



IN THE UNITED STATES PATENT AND TRADEMARK OFFICE

In re Patent Application of:

Akio MATSUDA, et al.

Application No.:

Group Art Unit:

Filed: September 28, 2001

Examiner:

For: METHOD OF SIMULATING OPERATION OF LOGICAL UNIT, AND COMPUTER-
READABLE RECORDING MEDIUM RETAINING PROGRAM FOR SIMULATING
OPERATION OF LOGICAL UNIT

**SUBMISSION OF CERTIFIED COPY OF PRIOR FOREIGN
APPLICATION IN ACCORDANCE
WITH THE REQUIREMENTS OF 37 C.F.R. § 1.55**

Assistant Commissioner for Patents
Washington, D.C. 20231

Sir:

In accordance with the provisions of 37 C.F.R. § 1.55, the applicant(s) submit(s) herewith
a certified copy of the following foreign application:

Japanese Patent Application No. 2000-403135

Filed: December 28, 2000

It is respectfully requested that the applicant(s) be given the benefit of the foreign filing
date(s) as evidenced by the certified papers attached hereto, in accordance with the
requirements of 35 U.S.C. § 119.

Respectfully submitted,

STAAS & HALSEY LLP

Date: September 28, 2001

By: 

James D. Halsey, Jr.
Registration No. 22,729

700 11th Street, N.W., Ste. 500
Washington, D.C. 20001
(202) 434-1500

日 本 国 特 許 庁
JAPAN PATENT OFFICE

JP997 U.S. PTO
09/964591
09/28/01

別紙添付の書類に記載されている事項は下記の出願書類に記載されている事項と同一であることを証明する。

This is to certify that the annexed is a true copy of the following application as filed with this Office

出 願 年 月 日

Date of Application:

2000年12月28日

出 願 番 号

Application Number:

特願2000-403135

出 願 人

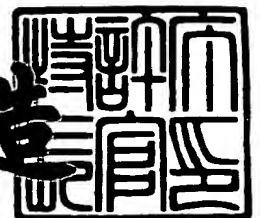
Applicant(s):

富士通株式会社

2001年 6月20日

特 許 庁 長 官
Commissioner,
Japan Patent Office

及 川 耕 造



出証番号 出証特2001-3058462

【書類名】 特許願

【整理番号】 0051937

【提出日】 平成12年12月28日

【あて先】 特許庁長官殿

【国際特許分類】 G06F 19/00

【発明の名称】 論理装置の動作シミュレーション方法並びに論理装置の
動作シミュレーションプログラムを記録したコンピュ
ータ読み取り可能な記録媒体

【請求項の数】 10

【発明者】

【住所又は居所】 神奈川県川崎市中原区上小田中4丁目1番1号 富士通
株式会社内

【氏名】 松田 明男

【発明者】

【住所又は居所】 神奈川県川崎市中原区上小田中4丁目1番1号 富士通
株式会社内

【氏名】 朱 強

【発明者】

【住所又は居所】 神奈川県川崎市中原区上小田中4丁目1番1号 富士通
株式会社内

【氏名】 松崎 和浩

【発明者】

【住所又は居所】 神奈川県川崎市中原区上小田中4丁目1番1号 富士通
株式会社内

【氏名】 土居 武史

【特許出願人】

【識別番号】 000005223

【氏名又は名称】 富士通株式会社

【代理人】

【識別番号】 100092978

【弁理士】

【氏名又は名称】 真田 有

【電話番号】 0422-21-4222

【手数料の表示】

【予納台帳番号】 007696

【納付金額】 21,000円

【提出物件の目録】

【物件名】 明細書 1

【物件名】 図面 1

【物件名】 要約書 1

【包括委任状番号】 9704824

【プルーフの要否】 要

【書類名】 明細書

【発明の名称】 論理装置の動作シミュレーション方法並びに論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体

【特許請求の範囲】

【請求項 1】 プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを、当該ハードウェアリソースを管理するリソースマネージャに要求するリソース要求ステップと、

該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、

該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを有するとともに、

該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステップを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションすることを特徴とする、論理装置の動作シミュレーション方法。

【請求項 2】 該リソースマネージャが、該リソース要求ステップによるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することを特徴とする、請求項 1 記載の論理装置の動作シミュレーション方法。

【請求項 3】 該リソースマネージャが、該リソース要求ステップによるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定することを特徴とする、請求項 1 記載の論理装置の動作シミュレーション方法。

【請求項 4】 該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該スレッドのボトルネックを検出することを特徴とする、請求項 1 記載の論理装置の動作シミュレーション方

法。

【請求項 5】 該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該リソース要求のブロッキングを検出することを特徴とする、請求項 1 記載の論理装置の動作シミュレーション方法。

【請求項 6】 該スレッドに、該リソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておくことを特徴とする、請求項 1 記載の論理装置の動作シミュレーション方法。

【請求項 7】 該スレッドに、該機能の実行制限時間を与えておくことを特徴とする、請求項 1 記載の論理装置の動作シミュレーション方法。

【請求項 8】 請求項 1 ～ 7 のいずれか 1 項に記載の動作シミュレーション方法によるシミュレーション結果と、該論理装置の動作予測値とを比較する比較ステップと、

該比較ステップでの比較結果を外部装置へ出力する出力ステップとを有することを特徴とする、論理装置の動作シミュレーション方法。

【請求項 9】 論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータを、

プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャ及び該スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとして機能させるとともに、

該スレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを該リソースマネージャに要求するリソース要求ステップと、該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを実行するとともに、該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステッ

プを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションするためのプログラムが記録されていることを特徴とする、論理装置の動作シミュレーションプログラムを記録した記録媒体。

【請求項 1 0】 論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータに、

請求項 1 ～ 7 のいずれか 1 項に記載の動作シミュレーション方法によるシミュレーション結果と該論理装置の動作予測値とを比較する比較ステップと、該比較ステップでの比較結果を外部装置へ出力する出力ステップとを実行させるための動作シミュレーションプログラムが記録されたことを特徴とする、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【発明の詳細な説明】

【 0 0 0 1 】

【発明の属する技術分野】

本発明は、L S I (Large Scale Integrated circuit) や I C (Integrated Circuit) などの論理装置を設計する上でその機能や性能の検証を目的として動作シミュレーションを行なうのに用いて好適な、論理装置の動作シミュレーション方法並びに論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体に関する。

【 0 0 0 2 】

【従来の技術】

従来、論理装置の設計は H D L (Hardware Description Language) を用いて R T (Register Transfer) レベルから設計を開始することが多い。しかし、R T レベルはクロックサイクル精度のハードウェアを意図した技術なので、ハードウェアとソフトウェアが明確に分離していない設計初期段階の論理装置を表現して機能を検証したり性能を評価したりすることは困難である。

【 0 0 0 3 】

そこで、R T レベルよりも抽象度の高い設計の初期段階で論理装置を記述する

言語〔例：C/C++, UML (Unified Modeling Language) , SDL (Specification and Description Language) など〕を用いる設計手法が数多く提案されている。例えば、以下に列記する文献がある。

- ① 「Hardware-Software Co-design of Embedded Systems - The POLIS approach」 (F.Balarin, E.Sentovich, M.Chiodo, P.Giusto, H.Hsieh, B.Tabbara and A.Jurecska, L.Lavagno, C.Passerone, K.Suzuki and A.Sangiovanni-Vincentelli, Kluwer Academic Publishers, 1997.)
- ② 「Coware - a design environment for heterogeneous hardware/software partitioning problem」 (K.Rompaey, D.Verkest, I.Bolsens and H.De Man, Proceedings EuroDAC, 1996.)
- ③ 「An Object Oriented Programming Approach for Hardware Design」 (S.Vernalde, P.Schaumont and I. Bolsens, IEEE Computer Society Workshop on VLSI, April, 1999.)
- ④ 「A Methodology and Design Environment for DSP ASIC Fixed Point Refinement」 (R.Cmar, L.Rijnders, P.Schaumont, S.Vernalde and I.Bolsens, Proceedings Design and Test in Europe Conference, pp. 271-276, March, 1999.)
- ⑤ 「Hardware Reuse at the Behavioral Level」 (P.Schaumont, R.Cmar, S.Vernalde, M.Engels and I.Bolsens, The proceedings of 36th Design Automation Conference. ACM, 1999, pp. 784-789, June 1999.)
- ⑥ 「An Efficient Implementation of Reactivity for Modeling Hardware in the SCENIC Design Environment」 (Stan Liao, Steve Tijang and Rajesh Gupta, Proceedings of the Design Automation Conference DAC'97, pp. 70-75, June 1999.)
- ⑦ 「SpecC System-Level Design Methodology Applied to the Design of a GSM Vocoder」 (A.Gerstlauer, S.Zhao, A.Horak and D.Gajski, Proceedings of the Workshop on Synthesis And System Integration of Mixed Technologies, April, 2000.)
- ⑧ 「On the use of C++ for system-on-chip design」 (Diederik Verkest, Joh

an Cockx and Freddy Potargent, Proceedings of the IEEE Workshop on VLSI (IWV), pp.42-47, April, 1999.)

【 0 0 0 4 】

これらの設計手法では論理装置の並列／並行処理を実現するためにソフトウェアのスレッド技術〔参考文献「並列オペレーティングシステム」（福田晃，（株）コロナ社，pp.30，1997.），「J a v a スレッドプログラミング」（Scott Oaks，Henry Wong共著，戸松豊和，西村利浩訳，（株）オーム社，1997.）など〕を導入している。

【 0 0 0 5 】

即ち、各スレッドを1つの処理ブロックにマッピングして、ブロック間には「通信」機能を定義することで論理装置のシミュレーションを実現している。

ここで、論理装置の設計コストに影響する大きな要素は二つ存在している。1つは手戻り作業の有無であり、もう1つは設計の再利用性である。

即ち、手戻り作業が多いほど、設計コストは増加する。特に、設計仕様書（以下、単に「仕様」という）からいきなりHDLを用いてRTレベルの設計を始める場合に、仕様の確認を行わず、実装設計を開始すると、実装した結果が要求されている仕様を満たさない場合に、仕様の見直しが必要となり、実装設計の再設計を行わなければならないケースが多い。これは設計工数、設計期間を大きく増大させる原因である。これを改善するために、早期に正確なパフォーマンスの見積もりを行なうことが重要である。

【 0 0 0 6 】

もう一方、設計の再利用性に関しては、二種類の再利用の可能性があげられる。一つは以前設計したモジュールの再利用であり、もう一つは設計を段階化したときの途中段階の記述（ソースコード）の再利用である。例えば、設計の途中からパフォーマンスを見積もった結果によって、再設計になった場合に、ソースコードを変更しなければならないことは設計コストに影響する。この視点から、仕様書から直接RTレベルを設計する手法は設計コストが高い。なぜなら、設計を変更すると、最悪の場合、全てのRTレベルでの記述を再記述しなければならないことがあるからである。

【 0 0 0 7 】

以上のように、仕様から直接 R T レベルで設計プロセスを開始することは設計コストの増大につながる事がわかる。そこで、これを解決するために、システムの上位設計の手法が提案されている。例えば図 3 0 (A) 及び図 3 0 (B) に示すような、段階的詳細化による設計アプローチである。

この段階的詳細化による設計アプローチは、概略して、以下のような手順によって設計が行なわれる。

【 0 0 0 8 】

・仕様 1 0 0 から機能（ブロック）を抽出してブロック図化する〔図 3 0 (A) のステップ A 1〕。そして、そのブロック図に基づいて、例えば、C 言語などによって、「UnTimed」レベルの機能モデル、つまり、時間概念を考慮しない機能モデルを構築（記述）する〔図 3 0 (A) のステップ A 2, 図 3 0 (B) のステップ B 1〕。なお、このとき、機能（ブロック）間に依存関係があれば、その間には「通信」機能が定義（記述）され、それらの機能（ブロック）はシーケンシャルに実行される。

【 0 0 0 9 】

・アーキテクチャの設計によって、ハードウェアにマッピングする機能ブロックの記述を B C A (Bus Cycle Accurate) に変更し〔図 3 0 (B) のステップ B 2〕、「通信」機能はバス通信プロトコルになる〔図 3 0 (B) のステップ B 3〕。この段階（レベル）でパフォーマンスの見積もりを行ない〔図 3 0 (A) のステップ A 3〕、仕様の確認を行なう。

【 0 0 1 0 】

・さらに記述を F C A (Full Cycle Accurate) に変更して〔図 3 0 (B) のステップ B 4〕、実際の R T レベルの記述に変更する〔図 3 0 (A) のステップ A 4〕。このとき、「通信」機能も記述に組み込む〔図 3 0 (A) のステップ A 5〕。

以上のような段階的な詳細化によって、早期に仕様の確認ができ、実行可能な仕様を実現することができる。

【 0 0 1 1 】

【発明が解決しようとする課題】

しかしながら、上述したような従来手法では、各レベルでの変更を全て手作業で行なうため、上記の「UnTimed」レベルからBCAレベルに変更する際に余計な設計コストが必要となる。さらに、その段階でのパフォーマンスの見積もり（上記のステップA3）において、設計の見直しが必要になった場合、記述を変更しなければならない。

【0012】

また、従来手法では、機能（ブロック）の構成にアーキテクチャが反映されているため、アーキテクチャの変更が必要になった場合、その機能（ブロック）の構成を変更しなければならず、これに伴ってその記述も変更しなければならなくなる。

【0013】

例えば図31に示すように、BCA記述により、ハードウェア資源（リソース）「A」を用いて機能「1」が実現（ハードウェア資源「A」に機能「1」がマッピングされる）とともに、ハードウェア資源「B」を用いて機能「2」が実現され、且つ、これらの機能「1」、「2」間に「通信」機能が定義された状態で、さらに、ハードウェア資源「A」を複数用いて上記と同じ機能「1」を追加実現する必要があった場合を想定する。

【0014】

この場合、追加分の機能「1」のBCA記述をハードウェア資源「A」の個数分だけ用意しなければならず、さらに、機能「1」と機能「2」との間の「通信」機能を既に定義済みの分も含めて再定義しなければならない。このように、従来手法では、機能（ブロック）の構成にアーキテクチャが反映されているため、アーキテクチャの変更が必要になる度に機能（ブロック）の記述を全て変更する必要があり、設計コストが大幅に増大し、設計効率が悪くなる。

【0015】

本発明は、このような課題に鑑み創案されたもので、論理装置の設計の初期段階において、動作シミュレーションを行なって、機能の確認、アーキテクチャの評価を行なえるようにするとともに、アーキテクチャの変更に対しても最小限の

記述の変更で柔軟に対応できるようにすることで、論理装置の設計コストを大幅に削減できるようにすることを目的とする。

【0016】

【課題を解決するための手段】

上記の目的を達成するために、本発明の論理装置の動作シミュレーション方法（請求項1）は、①プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャが、論理装置の設計仕様に応じてその論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを、そのハードウェアリソースを管理するリソースマネージャに要求するリソース要求ステップと、②上記のリソースマネージャが、その要求に応じたハードウェアリソースを予め規定されたルールに従って上記スレッドに割り当てるリソース割り当てステップと、③上記のスレッドマネージャが、このリソースマネージャによる割り当て結果に応じて上記スレッドの実行状態を制御するスレッド制御ステップとを有するとともに、上記スレッドの実行が完了するまで上記のスレッドマネージャとリソースマネージャとが連携して上記①～③の各ステップを繰り返し実行することにより、論理装置の動作完了までの動作をシミュレーションすることを特徴としている。

【0017】

なお、上記のリソースマネージャは、上記のリソース要求ステップ（①）によるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定するようにしてもよい（請求項2）。

【0018】

また、上記のリソースマネージャは、上記のリソース要求ステップ（①）によるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定するようにしてもよい（請求項3）。

【0019】

さらに、上記のリソースマネージャは、上記のハードウェアリソースに対する

リソース要求数を監視し、その監視結果に基づいてスレッドのボトルネックを検出するようにしてもよい（請求項４）。

また、上記のリソースマネージャは、上記のハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいてリソース要求のブロッキングを検出するようにしてもよい（請求項５）。

【 0 0 2 0 】

さらに、上記のスレッドには、上記のリソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておいてもよい（請求項６）、上記の機能に実行制限時間を与えておいてもよい（請求項７）。

また、本発明の論理装置の動作シミュレーション方法（請求項８）は、上記の動作シミュレーション方法によるシミュレーション結果と、上記論理装置の動作予測値とを比較する比較ステップと、この比較ステップでの比較結果を外部装置へ出力する出力ステップとを有していることを特徴としている。

【 0 0 2 1 】

次に、本発明の論理装置の動作シミュレーションプログラムを記録した記録媒体（請求項９）は、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、上記のコンピュータを、プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャ及び上記スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとして機能させるとともに、①上記のスレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを上記のリソースマネージャに要求するリソース要求ステップと、②上記のリソースマネージャが、その要求に応じたハードウェアリソースを予め規定されたルールに従って上記スレッドに割り当てるリソース割り当てステップと、③上記のスレッドマネージャが、上記のリソースマネージャによる割り当て結果に応じて上記スレッドの実行状態を制御するスレッド制御ステップとを実行するとともに、上記スレッドの実行が完了するまで上記のスレッドマネージャとリソースマネージャとが連携して上記①～③の各ステップを繰り返し実行

することにより、論理装置の動作完了までの動作をシミュレーションするためのプログラムが記録されていることを特徴としている。

【0022】

さらに、本発明の論理装置の動作シミュレーションプログラムを記録した記録媒体（請求項10）は、コンピュータに、上記の動作シミュレーション方法によるシミュレーション結果と論理装置の動作予測値とを比較する比較ステップと、この比較ステップでの比較結果を外部装置へ出力する出力ステップとを実行させるための動作シミュレーションプログラムが記録されたことを特徴としている。

【0023】

【発明の実施の形態】

以下、図面を参照して本発明の実施の形態を説明する。

（A）第1実施形態の説明

図1は本発明の第1実施形態に係るパーソナルコンピュータなどの計算機（情報処理装置）の構成を示すブロック図で、この図1に示す計算機1は、計算機本体2とディスプレイ（表示装置）3とをそなえており、計算機本体2（以下、単に「本体2」と略記することがある）には、例えば、CPU（Central Processing Unit）4、主記憶部（メモリ）5、二次記憶装置（ハードディスク）6、フロッピーディスク（FD）ドライブ7などがそなえられており、これらのコンポーネントがPCI（Peripheral Component Interconnect）バスなどの内部バス8を介して相互に通信可能に接続されている。

【0024】

ここで、上記のCPU4は、計算機1としての動作を統括制御するためのもので、メモリ5やハードディスク6に内部バス8を介してアクセスして必要なソフトウェア（アプリケーション）プログラム（以下、単に「プログラム」ともいう）やアプリケーションデータなどを読み込んで動作することによって、計算機1として必要な機能（本実施形態では、特に、論理装置の動作シミュレーション装置としての機能）が発揮されるようになっている。なお、その動作結果（論理装置の動作シミュレーション結果を含む）は、適宜、ディスプレイ3に表示（出力）される。

【0025】

また、ハードディスク6は、上記のプログラムやアプリケーションデータなど（以下、説明の便宜上、単に「各種データ」と総称することがある）を予め、あるいは、インストールなどによって記憶しておくためのもので、ここに記憶されている各種データが、適宜、CPU4からのアクセス速度が高速なメモリ5に読み出されて、CPU4によるプログラムの実行が高速に行なわれるようになっている。

【0026】

さらに、FDドライブ7は、フロッピーディスク（FD；記録媒体）9に記録されている各種データをCPU4の制御のもとに読み出してハードディスク6に記憶することによって、各種データのインストールを可能にする機能を提供するもので、例えば、FD9に、論理装置の動作シミュレーションプログラム10が記録されていれば、このFD9からその動作シミュレーションプログラム10をインストールする（ハードディスク6に記憶する）ことによって、計算機1（CPU4）を論理装置の動作シミュレーション装置として機能させることが可能である。

【0027】

なお、上記の論理装置の動作シミュレーションプログラム10（以下、単に「シミュレーションプログラム10」、もしくは、「プログラム10」という）がハードディスク6あるいはメモリ5に記憶された時点で、そのプログラム10を保持したハードディスク6あるいはメモリ5が上記シミュレーションプログラム10を記録した記録媒体となることはいうまでもない。

【0028】

また、このプログラム10は、勿論、FD9以外の記録媒体〔例えば、CD-ROMや光磁気ディスク（MO）など〕に記録されていてもよく、対応するドライブなどが本体2に装備されていれば、上記と同様に、それらの記録媒体からのインストールが可能である。さらに、記録媒体からのインストールだけでなく、例えば、インターネットなどの所望の通信回線（ネットワーク）を介したオンラインでのインストールも可能である。つまり、プログラム10は、FD9やCD

ーROM, MOなどの記録媒体として提供されてもよいし、インターネットなどの所望の通信回線を介して提供されてもよい。

【0029】

さて次に、以下では、上記のシミュレーションプログラム10の詳細について説明する。

本実施形態のシミュレーションプログラム10は、例えば図2に示すようなOMT (Object-oriented Modeling Technique) に基づいて記述 (オブジェクト指向プログラミング) されており、その要部に着目すると、この図2に示すように、スレッドマネージャ11, リソースマネージャ12, スレッド13, リソース14, テストベンチ15, テストデータ16, 入力キュー17, 実行待ちキュー18, (リソース) 要求19, 回答21, リソース回答キュー22などが定義されている。なお、「OMT」については、例えば、文献「オブジェクト指向方法論OMT」(J.ランボー, M.ブラハ, W.プレラニ, F.エディ, W.ローレンセン共著, 羽生田栄一訳, (株)トッパン, 1995.) になどに詳しく解説されている。

【0030】

ここで、まず、テストベンチ15は、論理装置の動作シミュレーションを行なうのに必要なテストデータ16を所定のデータ生成ルールに従って生成してから実行権利を入力キュー17に譲るという機能を提供するもので、このテストベンチ15で生成されたテストデータ16がテストベンチ15の所有する入力キュー17に順次格納されたのち、スレッドマネージャ11によって順次取り出されて、スレッド13の生成に使用されるようになっている。

【0031】

次に、上記のスレッド13は、「実行権利を譲る」、「実行する」、「停止する」、「停止する」、「再開する」、「開始する」、「消滅する」、「生成する」などの関数 (OMTでは「メソッド」と呼ばれる) が定義されることにより、スレッドマネージャ11から実行権利を譲り受けた場合にスレッド13の実行状態が制御されるようになっている。

【0032】

このスレッド13は、プログラムの実行単位を表し、本実施形態では、実現し

たい論理装置の設計仕様に応じて必要となる論理装置の入力から出力（動作完了）までの処理（機能）を表した（定義した）逐次フローとして記述される。仮に、論理装置の入力から出力までに必要な処理（機能）を処理A→処理B→処理Cとし、これらを1つのスレッド13に表すとする、例えばC++記述スタイルを用いた場合、図5に示すように記述される。なお、以降の説明においても、シミュレーションプログラム10の具体的な記述例については、特に断らない限り、C++記述スタイルを適用するものとする。

【0033】

そして、上記のスレッド13は、実行状態になると、その進行（即ち、上記の処理A、B、Cなどの「メソッド」の実行）のためにリソース14を確保する必要がある場合はそれを要求する要求（リソース要求）19を、確保したリソース14を解放する場合にはそれを要求する要求（解放要求）19を、それぞれ、自立的に発行してリソース要求キュー20に順次格納してゆくようになっている。

【0034】

なお、ここでいうリソース14とは、本実施形態では、ハードウェア資源（ハードウェアリソース）に関する情報を意味し、例えば、プロセッサ、ASIC（Application Specific Integrated Circuit）、演算器などがそれぞれリソースデータとして定義される。

【0035】

次に、実行待ちキュー18は、複数のキューのそれぞれに実行待ちのスレッド13を格納する機能を提供するもので、この実行待ちキュー18は、スレッドマネージャ11に集約されている。

【0036】

つまり、本実施形態のスレッドマネージャ11は、実行待ちキュー18と1つ以上のスレッド13から構成され、スレッド13の進行（生成、実行、停止、再開、消滅などの「メソッド」）及び複数のスレッド13の実行スケジュール（順序）を管理する（即ち、スレッド13の制御を行なう）タスクとして機能する。なお、スレッド13の次の動作は、このスレッドマネージャ11が、リソースマネージャ12の発行する回答21をリソース回答キュー22から取り出して参

照することで決定される（詳細については後述する）。

【0037】

さらに、リソースマネージャ12は、スレッド13が進行するのに（「メソッド」の実行に）必要なリソース14（種類及び個数）を管理し、予め規定された上記「調停ルール」に従って単位時間当たりのリソース要求19をリソース14の種類毎に調停しながら、リソース要求19に応じたリソース14を要求元のスレッド13に割り当てるものである。

【0038】

なお、このリソースマネージャ12によるリソース14の割り当て結果は回答21として発行され、一旦、リソース回答キュー22に格納（追加）されたのち、スレッドマネージャ11によって取り出されて参照される。また、上記の「単位時間」とは、リソースマネージャ12がリソース14の調停を行なう最小の時間単位を意味する。

【0039】

ここで、本リソースマネージャ12を、上記のスレッド13と同様にC++記述スタイルで記述すると、例えば図6に示すようになる。即ち、リソースマネージャ12が管理するリソース（R）14の種類を例えば“R1”，“R2”とし、上記のリソース要求キュー20とリソース回答キュー22とを定義する。

【0040】

そして、要求メソッド31a、リソース14の解放を行なう解放メソッド31b、リソース14の調停を行なう調停メソッド31cをそれぞれ定義する。ここで、要求メソッド31aは、どの種類のリソース14（“R1” or “R2”）を要求するのかをパラメータとして明確に指定し、要求したリソース14とその要求19を出したスレッド13のスレッドIDとを要求メッセージとしてリソース要求キュー20に登録する操作を行なうための関数である。

【0041】

また、解放メソッド31bは、リソース14の解放を行ない、解放したリソース14の個数を1つ増やす操作を行なう関数であり、調停メソッド31cは、リソース14の種類（“R1”，“R2”）毎にそれぞれの「調停ルール」に従っ

てリソース割り当ての調停を行なうための関数で、破線枠 3 1 1 内の記述により、リソース要求キュー 2 0 が空でない限り、リソース要求キュー 2 0 から 1 つずつ要求 1 9 が取り出され、例えば、リソース 1 4 の種類が“R 1”の場合は破線枠 3 1 2 及び 3 1 3 内の記述が実行され、リソース 1 4 の種類が“R 2”の場合は破線枠 3 1 4 及び 3 1 5 内の記述が実行される。

【 0 0 4 2 】

なお、破線枠 3 1 2, 3 1 4 内の記述は、要求されたリソース 1 4 (“R 1”もしくは“R 2”) の個数がその時点で 0 以下であれば、割り当てられるリソース 1 4 が無いので、リソース割り当て結果の回答 2 1 として“False”をリソース回答キュー 2 2 に格納することを意味する。

また、破線枠 3 1 3, 3 1 5 内の記述は、上記以外の場合、即ち、要求されたリソース 1 4 (“R 1”もしくは“R 2”) の個数が 1 以上であれば、割り当て可能なので、そのリソース 1 4 の調停ルールに従ってリソース 1 4 の割り当てを行なって、その調停(割り当て)結果(“True”)を回答 2 1 としてリソース回答キュー 2 2 に格納することを意味する。

【 0 0 4 3 】

以上のような構成により、本実施形態のシミュレーションプログラム 1 0 は、計算機 1 (CPU 4) を、次のように動作(機能)させることが可能である。

即ち、図 3 に模式的に示すように、まず、テストベンチ 1 5 が規定の「テスト生成ルール」に従ってテストデータ 1 6 を生成して(ステップ S 1)、入力キュー 1 7 に追加する(ステップ S 2)。

【 0 0 4 4 】

テストベンチ 1 5 は、全てのテストデータ 1 6 を生成した後、スレッドマネージャ 1 1 に実行権利を譲る(ステップ S 3)。実行権利を譲り受けたスレッドマネージャ 1 1 は、入力キュー 1 7 からテストデータ 1 6 を取り出して(ステップ S 4)、それに対応するスレッド 1 3 を生成し(ステップ S 5; スレッド生成ステップ)、実行待ちキュー 1 8 に追加する(ステップ S 6)。なお、この時点でのスレッド 1 3 は何も実行していない未開始状態にある。

【 0 0 4 5 】

そして、スレッドマネージャ11は、入力キュー17が空になってから実行待ちキュー18から実行待ちのスレッド13を取り出し、そのスレッド13が未開始状態であれば実行させて「実行状態」にする（ステップS7）。「実行状態」となったスレッド13は、リソースマネージャ12に対して最初の「メソッド」に必要なリソース14の確保を要求する（例えば、リソース「A」～「C」のうちのリソース「B」に対するリソース要求19を発行する；ステップS8；リソース要求ステップ）。このリソース要求19は、スレッドマネージャ11によって、リソース要求キュー20に追加される。また、このリソース要求19を発行したスレッド13は、実行待ちキュー18の最後に追加される。

【0046】

その後、スレッドマネージャ11は、実行待ちキュー18に格納されている全てのスレッド13の処理（実行）が終了すると、実行権利をリソースマネージャ12に譲る（ステップS9）。実行権利を譲り受けたリソースマネージャ12は、リソース要求キュー20の先頭から、順次、リソース要求19を取り出して（ステップS10）、現在の「調停ルール」に従ってリソース14の割り当て調停を行ないながら、要求されたリソース14のスレッド13への割り当てを行ない、その結果（割り当てられたか否か；“True” or “False”）を回答21として発行し、リソース回答キュー22に追加してゆく（ステップS11；リソース割り当てステップ）。

【0047】

そして、リソース要求キュー20に格納されている全ての要求19に対するリソース割り当て処理が終了してから、リソースマネージャ12は、実行権利をスレッドマネージャ11に譲る（ステップS12）。実行権利を譲り受けたスレッドマネージャ11は、リソース回答キュー22から回答21を、順次、取り出して（ステップS13）、その回答21の内容が“True”（割り当てOK）であれば、スレッドIDに対応するスレッド13（この時点で、実行待ちキュー18の先頭に位置している）を実行させて（スレッド制御ステップ）、次の「メソッド」の実行に必要なリソース要求19を発行し、これをリソース要求キュー20に追加してから、実行したスレッド13を再び実行待ちキュー18の最後に追加す

る。

【0048】

なお、リソース回答キュー22から取り出した回答21が“False”（割り当てNG）であった場合、スレッドマネージャ11は、現在の（前回発行した）リソース要求19を再びリソース要求キュー20に追加してから、スレッド13を、「待ち状態」として再度、実行待ちキュー18の最後に追加する（スレッド制御ステップ）。また、全ての処理を終えた（リソース14の割り当てが完了した）スレッド13は消滅させて、実行待ちキュー18からそのスレッド13を削除する。

【0049】

以上のようにして、スレッドマネージャ11とリソースマネージャ12とが連携して、全てのスレッド13の実行（全ての「メソッド」の実行）が完了するまでリソース要求、リソース割り当て及び回答21に応じたスレッド13の実行状態の制御を繰り返し実行することによって、論理装置の動作シミュレーションを実施する。

【0050】

つまり、本実施形態のスレッドマネージャ11及びリソースマネージャ12は、それぞれ、図4に模式的に示すように、以下の各機能部を有しており、スレッド13の実行が完了するまで、これらのスレッドマネージャ11とリソースマネージャ12とが連携して上記のリソース要求19及びスレッド13の実行状態の制御を繰り返し実行して、その実行結果を例えばディスプレイ3に出力することによって、論理装置の動作完了までの動作をシミュレーションするようになっているのである。

【0051】

スレッドマネージャ11

- ・テストデータ16の入力により、論理装置の設計仕様に応じてその論理装置の動作完了までに必要となる機能（処理）を表したスレッドを生成するスレッド生成手段110としての機能

- ・スレッド13の実行に必要なリソース14をリソースマネージャ12に要求

するリソース要求手段 1 1 1 としての機能

・このリソース要求手段 1 1 1 による要求 1 9 に対するリソースマネージャ 1 2 による割り当て結果（回答 2 1）に応じてスレッド 1 3 の実行状態を制御するスレッド制御手段 1 1 2 としての機能

リソースマネージャ 1 2

・リソース要求 1 9 に応じたリソース 1 4 を予め規定された「調停ルール」に従ってスレッド 1 3 に割り当てるリソース割り当て手段 1 2 1 としての機能

【 0 0 5 2 】

換言すれば、本実施形態では、論理装置に実装すべき「処理（機能）」が必要とするハードウェア資源に関する記述はスレッド 1 3（「メソッド」）には行なわず、そのスレッド 1 3 の「メソッド」が実行されるときに、その都度、必要なリソース 1 4 がリソースマネージャ 1 2 によって動的に割り当てられるようになっているのである。つまり、従来のように実現すべき「機能」にアーキテクチャは反映されていないのである。

【 0 0 5 3 】

従って、論理装置の設計初期段階における動作シミュレーションが実現できて設計初期段階において、仕様によって要求される「機能」を正確に実現できているかといった機能（アルゴリズム）の検証が行なえるとともに、アーキテクチャに変更が必要になった場合でも、「機能」の記述変更は殆ど行なわずに、リソース 1 4 やリソース 1 4 の「調停ルール」，スレッド 1 3 のスケジューリング，リソース 1 4 に対する要求などを、その変更に応じて変更するだけで、簡単に設計の再検証を行なうことが可能になる。

【 0 0 5 4 】

なお、このようにリソース 1 4 やリソースの「調停ルール」，スレッドのスケジューリング，リソースに対する要求などを変更するだけで設計の再検証を行なえるということは、「機能」に適したアーキテクチャの探索や、「機能」をアーキテクチャへマッピングする際のアーキテクチャの性能評価，設計初期段階におけるタイミングの検証なども行なえることを意味する。

【 0 0 5 5 】

(A 1) 第 1 実施形態の具体例の説明

さて次に、以下では、上述したシミュレーション方法について、具体例を挙げて、より詳細に説明する。即ち、ここでは、QoS (Quality of Service) システムの設計とその動作シミュレーションを行なう場合について説明する。なお、ここでいう「QoS システム」とは、IP (Internet Protocol) ルータなどのパケット転送装置において、例えば図 7 に模式的に示すように、入力 (受信) パケット 41 を或るルールに従ってクラス分けして複数のパケットキュー 45 に格納し、最終的に、これらの各キュー 45 から規定のサービスレートを満足するよう [規定のサービスレートに応じた重み付け (W0 ~ W3 など) に応じて] パケット 41 を出力するというものである。「QoS」の詳細については例えば、文献「インターネット QoS」(Paul Ferguson, Geoff Huston 共著, 戸田 巖 監訳, (株)オーム社, 2000.) に記載されている。

【0056】

①仕様からの機能の抽出

まず、最初のステップとして、図 8 に示すように、仕様 40 から実現すべき QoS システムの「機能」を抽出する必要がある (ステップ S20)。即ち、この場合は、以下の (a) ~ (c) に示す機能が最低限必要になる。

【0057】

(a) パケット 41 のクラス分け機能 42 (図 7 参照) : Classify()

この機能 42 は、パケットヘッダの「IP Precedence フィールド」の値によってクラス分けする機能で、本実施形態では、例えば次表 1 に示すルールに従ってクラス分けを行なうものと仮定する。

【0058】

【表 1】

クラス分け

IP Precedence	クラス	対応するキュー番号
000	0	0
001		
010	1	1
011		
100	2	2
101		
110	3	3
111		

【0059】

(b) パケットをキュー45に格納する(キューイング)機能43: Queuing()

この機能43は、上記のクラス分け機能42によってクラス分けされたパケット41を対応するキュー45に格納する機能である。ここで、各キュー45には格納できる最大パケット数(以下、最大サイズという)が予め決められており、最大サイズを超える分のパケット41については廃棄されるものとする。

【0060】

(c) キュー45からパケットを取り出す(デキュー)機能44: Dequeueing()

このデキュー機能44は、各キュー45からパケット41を取り出す(出力する)機能であるが、各キュー45には予め「サービスレート」が定められており、この「サービスレート」によってキュー45からパケット41を出力するか否かを決定する。

【0061】

②「機能」のスレッド化

次に、上記の各機能42～44を「メソッド」としてスレッド化する(図8の

ステップS21)。即ち、前記の処理A→処理B→処理Cという一連の処理（機能）を、図5により前述したように次のような1つのスレッド13として表す。

Classify() → Queuing() → Dequeuing()

ここで、このスレッド13の状態は次表2に示すように3種類とする。

【0062】

【表2】

スレッド状態

スレッド状態	説明
実行状態	スレッドが実行されている状態
待ち状態	スレッドが事象を待っている状態
終了状態	スレッドが終了されている状態

【0063】

なお、上記の各状態はリソース14の確保状況（割り当て結果）、キューイング機能42 [Queuing()]、デキュー機能43 [Dequeuing()]の結果に依存する。例えば、Queuing()を実行しようとしたときに、キュー45が既にいっぱいになっていて、パケット41を廃棄しなければならない場合には、スレッド13は「実行状態」から「終了状態」に変わる。また、リソース14が確保できない場合は、そのスレッド13は図3により上述したように実行待ちキュー18の最後に追加されて「待ち状態」となる。

【0064】

③仕様40からのアーキテクチャの設計

さて、上記の一方で、仕様40を基にQoSシステムのアーキテクチャを設計する必要がある（ステップS22）。即ち、例えば、上記のClassify()とQueuing()とを或るリソース14（仮に、R1とする）の上に実現し、Dequeuing()を別のリソース14（仮に、R2とする）の上に実現する。

【0065】

ここで、上記のClassify()がリソースR1上に実現されているとき〔つまり

、Classify()がリソースR 1を確保して実行中]の実行遅延時間を T_{1c} 、同様に、Queuing()がリソースR 1上で実現されているときの実行遅延時間を T_{1q} 、Dequeuing()がリソースR 2上で実現されているときの実行遅延時間を T_{2d} とする。

【0 0 6 6】

また、リソースR 1の競合が発生した場合は、Classify()からの要求を優先し、リソースR 2の競合が発生した場合は、要求の到着順位でリソースを割り当てるものとする。さらに、上記Dequeuing()の性能をアップするため、リソースR 2の数を例えば2つとする。

【0 0 6 7】

④アーキテクチャからのリソースの抽出

以上により、リソース1 4 (R 1, R 2)の個数及び調停ルールが次表3に示すように定義される(図8のステップS 2 3)。

【0 0 6 8】

【表3】

リソース一覧

リソース名	個数	調停ルール
R 1	1	Classify()が優先される。
R 2	2	先着要求が優先される。

【0 0 6 9】

④スレッド1 3へのリソース要求の取り込み

次に、上記のClassify(), Queuing()およびDequeuing()を実行するために、上記のリソースR 1とR 2を確保しなければならない(スレッド1 3にリソース要求を組み込む; 図8のステップS 2 4)。即ち、以下の手順(1)～(5)を実現(記述)する必要がある。

【0 0 7 0】

(1) パケット4 1が到着したら、スレッド1 3が動的に生成され、スレッド

1 3 が実行状態になる。

(2) Classify() を実行するために、リソース R 1 を確保しようとする (リソース R 1 についての要求 1 9 を発行する)。確保できたら、次へ進む [Classify() を実行する]。確保できなかったら、そのスレッド 1 3 は「待ち状態」になる。次のステップで (次に、スレッド 1 3 が「実行状態」となったときに)、再度、リソース R 1 の確保を行なう。

【 0 0 7 1 】

(3) Classify() を実行した後、Queuing() を実行するために、リソース R 1 を確保しようとする。確保できたら、次へ進む [Queuing() を実行する]。確保できなかったら、スレッド 1 3 は「待ち状態」となり、次のステップで (次にスレッド 1 3 が「実行状態」となったときに)、再度、リソース R 1 の確保を行なう。

【 0 0 7 2 】

(4) Queuing() を実行した後、Dequeuing() を実行するために、リソース R 2 を確保しようとする。確保できたら、次へ進む [Dequeuing() を実行する]。確保できなかったら、スレッド 1 3 は「待ち状態」となり、次のステップで (次にスレッド 1 3 が「実行状態」となったときに)、再度、リソース R 2 の確保を行なう。

(5) 全ての処理が終了したので、スレッド 1 3 を終了させる。

【 0 0 7 3 】

⑥ リソース調停ルールの記述

次に、複数のスレッド 1 3 が同じリソース 1 4 (R 1 又は R 2) に対して要求した場合、リソース 1 4 の競合が起きるため、「調停ルール」を定義する必要がある (図 8 のステップ S 2 5)。

【 0 0 7 4 】

まず、リソース R 1 の「調停ルール」 (図 6 に示した破線枠 3 1 3 内における「R 1 の調停ルール」に相当) は以下の手順が実現されるように記述する。

(1) 現在、リソース R 1 が空いているかどうかを判別する。空いていなければ、全ての要求 1 9 に対して、割り当てられない旨の回答 2 1 を発行する。空い

ていれば、次へ進む。

【0075】

(2) 単位時間内の要求19の中にClassify()を実行するためのリソース要求があれば、そのスレッド13 (複数ある場合は最先に要求19を出したスレッド13) にリソースR1を割り当てる。Classify()を実行するためのリソース要求19がなければ、最先に要求19を出したスレッド13にリソースR1を割り当てる。それ以外の要求19に対しては割り当てられない旨の回答21を発行する。

【0076】

一方、リソースR2の「調停ルール」(図6に示した破線枠314内における「R2の調停ルール」に相当) は以下の手順が実現されるように記述する。

(1) 現在、リソースR2が空いているかどうかを判別する。空いていなければ、全ての要求19に対して、割り当てられない旨の回答21を発行する。空いていれば、次へ進む。

【0077】

(2) 単位時間内の要求19の中で最先に要求を出したスレッド13にリソースR1を割り当てる。それ以外の要求19に対しては割り当てられない旨の回答21を発行する。

【0078】

⑦シミュレーション

以上のようにしてQoSシステムに必要な「機能」をプログラミングし(シミュレーションプログラム10を構築し)、計算機1(CPU4)上で実行させることで、QoSシステムの動作シミュレーションを実施する(図8のステップS26)。即ち、計算機1(CPU4)が、図3及び図4により前述したように、スレッドマネージャ11及びリソースマネージャ12として動作することにより、QoSシステムの動作シミュレーションが実施される。

【0079】

⑧シミュレーション結果と予測値の比較

上記のシミュレーションにより或るシミュレーション結果が得られる。即ち、

例えば、パケット 4 1 が或るレートで出力されることが分かる。そして、このシミュレーション結果と、仕様 4 0 から予測される動作結果（予測値）4 0' とを比較する（図 8 のステップ S 2 7 ; 比較ステップ）ことにより、設計したアルゴリズムが正しい（ステップ S 2 7 の Y E S ルートからステップ S 2 8）か否（設計ミス）か（ステップ S 2 7 の N O ルートからステップ S 2 9）が判断され、その結果が、例えば外部装置としてのディスプレイ 3 などに出力（表示）される（出力ステップ）。

【 0 0 8 0 】

つまり、図 8 に破線枠 5 1 内に示す部分は、上述した論理装置のシミュレーション方法によるシミュレーション結果と、その論理装置の動作予測値とを比較する比較ステップ（S 2 7）と、この比較ステップ（S 2 7）での比較結果をディスプレイ 3 などの外部装置へ出力する出力ステップとを、計算機 1（CPU 4）に実行させて、論理装置の機能検証を行なうための（動作シミュレーション）プログラムとして機能する。

【 0 0 8 1 】

なお、本プログラム 5 1 も、例えば F D 9 や C D - R O M, M O などの記録媒体に記録することができ、その記録媒体に記録されたプログラム 5 1 を計算機 1 にインストールする（ハードディスク 6 に記憶する）ことで、計算機 1（CPU 4）を上述のごとく動作させることが可能になる。

【 0 0 8 2 】

そして、設計ミス（例えば、廃棄されてしまうパケット 4 1 の数が規定よりも多いなど）であれば、キュー 4 5 のサイズを変える〔キュー 4 5（リソース 1 4）に関する記述を変更する〕などして、最適なアーキテクチャが得られるまで、上記のシミュレーションを行なって、Q o S システムに最適なアーキテクチャを探索する。

【 0 0 8 3 】

なお、上記の予測値は、上記の例の場合、前述した Classify() の実行遅延時間 T_{1c} , Queuing() の実行遅延時間 T_{1q} および Dequeueing() の実行遅延時間 T_{2d} に基に算出できる。また、上記のシミュレーション結果や予測値との比較結果は、

ディスプレイ3だけでなく、プリンタなどの周辺機器に出力することも可能である。

【0084】

以上のようにして、論理装置としてのQoSシステムの機能検証（システムとしての要求を満足しているか否かの検証）を、システムの設計初期段階で実施することができる。また、アーキテクチャに変更が必要になった場合でも、使用するリソース14（R1，R2）やリソース14の「調停ルール」，スレッド13のスケジューリング，リソース14に対する要求などを、その変更に応じて変更するだけで、簡単に設計（アルゴリズム）の再検証を行なうことができる。従って、手戻り、記述スタイルの変更が少ない柔軟性の高い設計手法（つまり、再利用性に優れた記述スタイル）を実現することが可能となり、システムの設計コストを大幅に削減することができる。

【0085】

（B）第1変形例の説明

ところで、上述したリソースマネージャ12には、例えば図9に示すように、デッドロック検出機構122を追加してもよい。このデッドロック検出機構122は、複数のスレッド13がリソース14を取り合うことによって「デッドロック」に陥る可能性があるか否かを検出するためのものである。

【0086】

なお、ここでいう「デッドロック」とは、2つ以上のスレッド13が互いに相手が占有しているリソース13を要求して解決しない特定の状態になっていることを意味する。「デッドロック」については、例えば、文献「現代オペレーティングシステムの基礎」（萩原宏，津田孝夫，大久保英嗣共著，（株）オーム社，pp.63，1995.）で説明されている。

【0087】

例えば、それぞれ図10に示すように記述されたスレッド13が2つ（仮に、スレッド“1”，“2”とする）存在していた場合を考える。ただし、ここでのリソース14（“A”）の数は3個、リソース14（“B”）の数は2個と仮定する。また、この図10において、「リソースA. 要求（）」などの関数（括

弧) 内に表示された数値 (この場合は “2”) はスレッド ID を表し、以降の説明においても同様とする。

【0088】

そして、このような場合、各スレッド “1”, “2” のそれぞれのリソース割付けグラフ (resource allocation graph) は、図 11 (A) に示すようになる。

この図 11 (A) に示す状態を簡約 (reduce) して表すと、図 11 (B) に示すような状態となるが、この状態で既にそれ以上の簡約が不可能 (irreducible) な状態であることが分かる。

【0089】

これは、スレッド “1” とスレッド “2” との間で「デッドロック」が発生する可能性があることを意味する。デッドロック検出機構 122 は、このようにリソース割付けグラフを基に「デッドロック」の発生可能性を判断する。

こうすることで、設計の初期段階において「デッドロック」の発生可能性を検出することが可能となり、論理装置の機能検証を行なうことが可能となる。

【0090】

(C) 第 2 変形例の説明

さて、上記のリソースマネージャ 12 には、例えば図 12 に示すように、リード／ライト検出機構 123 を追加してもよい。ここで、このリード／ライト検出機構 123 は、次のような事象 (1) ～ (3) を監視 (検出) することにより、リード／ライトエラーの発生可能性を検出するためのものである。

【0091】

(1) 同一リソース 14 に対するリード要求とライト要求の同一単位時間内の発生

(2) 既にライト要求によって占有されているリソース 14 に対するリード要求の発生

(3) 既にリード要求によって占有されているリソース 14 に対するライト要求の発生

【0092】

つまり、このリード／ライト検出機構 1 2 3 の検出結果をみれば、複数のスレッド 1 3 がリソース 1 4 に対してシーケンシャルに書き込み動作もしくは読み出し動作を行なう時に、その順序が正しいか否かを判定（つまり、複数スレッド 1 3 間のリソース 1 4 に対するリード／ライト動作の競合状態を判定）して、設計した論理装置が正しく動作するか否かを検証することができるのである。

【0 0 9 3】

これを実現するには、要求 1 9 の記述を例えば図 1 3 に示すように拡張する。即ち、次表 4 に示すような意味をもつ「リード／ライトフラグ」を定義する。

【0 0 9 4】

【表 4】

リード／ライトフラグ＝0	リード／ライト要求を考慮しない
リード／ライトフラグ＝1	リード要求
リード／ライトフラグ＝2	ライト要求

【0 0 9 5】

加えて、リソース 1 4 の記述を例えば図 1 4 に示すように拡張する。即ち、リソース 1 4 の解放が行なわれた場合には、必ず「カレントフラグ (CurrentFlag)」を“0”にするようにしておき（破線枠 3 1 6 参照）、この「カレントフラグ」が“0”でないときに、要求 1 9 の「リード／ライトフラグ」とリソース 1 4 側の「カレントフラグ」とが一致しなければ、「リード／ライトエラーの発生可能性がある」といったエラー情報がディスプレイ 3 などに出力される（破線枠 3 1 7 参照）ようにするのである。

【0 0 9 6】

（D）第 3 変形例の説明

さらに、上記のリソースマネージャ 1 2 には、例えば図 1 5 に示すように、ボトルネック検出機構 1 2 4 を追加してもよい。ここで、このボトルネック検出機構 1 2 4 は、スレッド 1 3 の実行結果やリソース 1 4 に対するアクセス状況を基

に「ボトルネック」の検出を行なうためのもので、例えば、リソース 1 4 の個数と種類を制限した環境下でシミュレーションを行なって、各リソース 1 4 に対するアクセス回数やリソース 1 4 を割り当てられなかった時の待ち時間などを確認することによって、論理装置の「ボトルネック」となっている部分を検出することが可能となる。

【 0 0 9 7 】

具体的には、例えば図 1 6 に示すように、リソース 1 4 の記述に、要求の数（アクセス数）を計数する「メソッド」を加えるとともに、その計数結果を「要求合計メンバ関数〔要求合計（ ）〕」によって呼び出せるようにしておけば、リソース 1 4 のアクセス回数を監視する仕組みがリソースマネージャ 1 2 において実現される。

【 0 0 9 8 】

つまり、ボトルネック検出機構 1 2 4 によって、各リソース 1 4 の「要求合計メンバ関数」を呼び出せば、それまでの各リソース 1 4 に対するアクセス頻度を調べることができ、例えば、アクセス頻度が最も多いリソース 1 4 は論理装置の「ボトルネック」になっている可能性があると推測することができる。従って、論理装置の設計初期段階で論理装置の性能検証（ボトルネックの有無の検証）を行なうことができる。

【 0 0 9 9 】

（E）第 4 変形例の説明

また、上記のリソースマネージャ 1 2 には、例えば図 1 7 に示すように、ブロッキング検出機構 1 2 5 を追加してもよい。ここで、このブロッキング検出機構 1 2 5 は、スレッド 1 3 からのリソース 1 4 の要求 1 9 が所定の制限数を超えた場合の「ブロッキング」の発生状況を調べる機能を提供するもので、リソース 1 4 に例えば図 1 8 に示すような記述を加えることで実現される。

【 0 1 0 0 】

そして、リソース 1 4 の「個数」を制限した（予め定めておく）環境下でシミュレーションを行なうことで、割り当てられるリソース数を超える要求 1 9 があった場合〔if 文の条件（「個数」＜ 0）を満足する場合〕には「ブロッキング」

が発生したと考えられ、例えば「リソース A の要求はブロッキングする必要がある。」といったエラー情報をディスプレイ 3 などに出力する。

【0101】

このようにして、本変形例では、ブロッキング検出機構 125 により、リソース 14 の「個数」を制限した環境下でのスレッド 13 からのリソース要求が制限数を超えた場合のブロッキングの発生可能性を調べることで、論理装置の性能を評価することができる。

【0102】

なお、上述した第 1～第 4 変形例におけるデッドロック検出機構 122，リード／ライト検出機構 123，ボトルネック検出機構 124，ブロッキング検出機構 125 は、論理装置の検証項目に応じて、全てあるいは一部の組み合わせでそなえられていてもよい。

【0103】

(F) 第 5 変形例の説明

ところで、上記のスレッド 13 には、そのスレッド 13 に記述されている一連の処理（「機能」；「メソッド」）に必要な実行時間の見積もり値（つまり、リソースマネージャ 12 によって割り当てられたリソース 14 を占有する時間についての予算）である「タイムバジェット」を各「機能」に対してそれぞれ設定してもよい。

【0104】

即ち、図 8 により前述したようにアーキテクチャからリソース 14 を抽出して各リソース 14 に処理をマッピングする際に（ステップ S24 にてスレッド 13 にリソース要求を組み込んだ後）、各処理の見積もり時間（タイムバジェット）をスレッド 13 に追加する（図 19 のステップ S25'）。

【0105】

例えば図 20 に示すように、スレッド 13 に記述される「機能」を処理 1，処理 2 とすると、それぞれに対して delay(処理 1 のタイムバジェット)，delay(処理 2 のタイムバジェット)を設定するのである。

そして、仕様 40 から論理装置が n 個の入力データを処理する際の実行時間の

制限をTとし（図19のステップS26b）、テストベンチ15からn個のデータを生成させて、前述したごとくシミュレーションを実行する（ステップ26a）。

【0106】

その後、全てのスレッド13が消滅されるまでの時間tを計測し、その計測時間tが制限時間T内であれば正しい設計と判断できる（図19のステップS27'のYESルートからステップS28'）。逆に、計測時間tが制限時間Tよりも大きい場合には仕様40の性能要件を満たしていないと判断できる（図19のステップS27'のNOルートからステップS29'）。

【0107】

このようにして、各種「機能」の組み合わせで構成される論理装置が、仕様40により要求される性能要件を満たしているか否かを、設計の初期段階にて確認することができる。

【0108】

（G）第6変形例の説明

なお、上記のスレッド13には、例えば図21に示すように、上記のステップS24にてスレッド13にリソース要求を組み込む前に、図22に示すようにして「ライフタイム」を設定してもよい（ステップS21'）。なお、ここでいう「ライフタイム」とは、「機能」の実行制限時間を表す。

【0109】

そして、第5変形例にて上述したようにスレッド13に「タイムバジェット」を設定（ステップS25'）した上で、シミュレーションを実行し（ステップS26''）、スレッド13が消滅する前にJudgeLifeTime()関数（図22参照）を（例えばリソースマネージャ12から）呼び出せば、上述した「タイムバジェット」による論理装置の性能検証に加えて、そのスレッド13に設定した「ライフタイム」が切れたか否か、つまり、「ライフタイム」内に処理が完了したか否かを判定することができる（ステップS27''）。

【0110】

この判定の結果、「ライフタイム」内にそのスレッド13の処理が完了してい

れば、論理装置が仕様40により要求されるリアルタイム性に関する性能要件を満足している（正しい設計である）と判断でき（ステップS27”のYESルートからステップS28”）、逆に、「ライフタイム」内に処理が完了していなければ、リアルタイム性に関する性能要件を満足していないと判断できる（ステップS29”）。

【0111】

このように、本変形例では、スレッド13に「タイムバジェット」と「ライフタイム」とを設定することで、各種「機能」の組み合わせで成る論理装置全体としての処理時間に関する性能要件とリアルタイム性に関する性能要件とを、論理装置の設計初期段階において検証することが可能となる。

なお、本変形例では、「タイムバジェット」と「ライフタイム」の双方を設定しているが、勿論、「ライフタイム」のみを設定して、論理装置のリアルタイム性のみを検証するようにしてもよい。

【0112】

また、これらの「タイムバジェット」、「ライフタイム」は、上述した例ではスレッド13に記述しているが、リソース14に記述することも可能である。例えば、メモリやバッファなどの記憶素子でのデータ保持時間が規定されるケースであれば、その記憶素子（リソース）に対して「タイムバジェット」や「ライフタイム」もしくはその両方を記述してもよい。

【0113】

（H）第7変形例の説明

上述した実施形態では、論理装置の動作完了までに必要な一連の処理（機能）を1つのスレッド13に表した場合について説明したが、上記「一連の処理」は、例えば図23に模式的に示すように、複数の逐次的なスレッド13（スレッド“1”～スレッド“n”）に表されていてもよい。

【0114】

この場合の動作は次のようになる。即ち、外部入力（テストデータ16の入力）によりスレッドマネージャ11が最初のスレッド“1”を生成する。すると、スレッド“1”は処理“1”を実行して、処理“1”に必要なリソース14をリ

ソースマネージャ 12 に要求する。

【0115】

これにより、スレッド “1” にリソース 14 が割り当てられてスレッド “1” が処理 “1” を完了し、スレッドマネージャ 11 によって消滅させられる際に、次のスレッド “2” を生成する。なお、リソース 14 が割り当てられなかった場合は、この場合も、そのスレッド 13 はスレッドマネージャ 11 によって「待ち状態」に制御される。

【0116】

以降、スレッド “i” (ただし、 $i = 1 \sim n - 1$) が消滅する際にそのスレッド “i” が次のスレッド “i + 1” を生成する。このようにして、論理装置の入力から動作完了までの一連の処理を一連の逐次的なスレッド 13 に表現する。つまり、本変形例は、スレッドマネージャ 11 が行なうスレッド 13 の生成を、スレッド 13 からも生成できるようにしたものである。

【0117】

具体的に、これを実現するには、例えば図 24 に示すように、スレッド 13 に対して「終了を待つ()」と言う「メソッド」を追加するとともに、処理 “1”，処理 “2”，処理 “3”，…に対応したスレッド “1”，スレッド “2”，スレッド “3”，…を追加する。

【0118】

これにより、各処理 “1”，“2”，“3”，…に対して 1 つずつスレッド 13 が生成され、それぞれのスレッド 13 の生成タイミングは前のスレッド 13 が処理を終了した後になる。これにより、スレッド 13 間の依存関係が前述した実施形態よりもさらに明確になる。

【0119】

そして、このように構成したスレッド 13 を図 3 により前述したシステムに組み込んで設計の初期段階における論理装置のシミュレーションを行なう。即ち、この場合も、全てのスレッド 13 の処理が完了（全てのスレッド 13 が消滅）するまで、スレッドマネージャ 11 とリソースマネージャ 12 とが連携して、リソース 14 の要求、リソース 14 の割り当て及びリソース 14 の割り当て結果に応

じたスレッド 1 3 の実行状態の制御を繰り返すことによって、論理装置の動作シミュレーションを実現する。

【 0 1 2 0 】

このように、本変形例では、論理装置の入力から動作完了までの一連の処理を一連の逐次的なスレッド 1 3 に表現して、シミュレーションを行なうので、論理装置に必要な処理の依存関係を明確にした上で、その機能検証を設計初期段階において行なうことができる。

【 0 1 2 1 】

(I) 第 8 変形例の説明

なお、上記の論理装置の動作完了までの一連の処理は、例えば図 2 5 に模式的に示すように、複数の逐次的または並行に実行されるスレッド 1 3 で表してもよい。即ち、第 7 変形例にて上述したようにスレッド 1 3 が消滅する際に、そのスレッド 1 3 が複数のスレッド 1 3 を同時に生成して各スレッド 1 3 を並列に実行させ、これら複数のスレッド 1 3 が消滅する際に、さらに次のスレッド 1 3 を逐次的に生成するようにしてもよい。

【 0 1 2 2 】

この場合の動作は次のようになる。即ち、外部入力（テストデータ 1 6 の入力）によりスレッドマネージャ 1 1 が最初のスレッド “ 1 ” を生成する。スレッド “ 1 ” は処理 “ 1 ” を実行して、処理 “ 1 ” に必要なリソース 1 4 をリソースマネージャ 1 2 に要求する。これにより、スレッド “ 1 ” にリソース 1 4 が割り当てられてスレッド “ 1 ” が処理 “ 1 ” を完了すると、スレッドマネージャ 1 1 によって消滅させられる際に、次のスレッド “ 2 ” ～ “ n ” を同時に生成する。なお、リソース 1 4 が割り当てられなかった場合は、この場合も、そのスレッド 1 3 はスレッドマネージャ 1 1 によって「待ち状態」に制御される。

【 0 1 2 3 】

その後、各スレッド “ 2 ” ～ “ n ” が並行して処理を進めてそれぞれの処理が完了し、これらの各 “ 2 ” ～ “ n ” がスレッドマネージャ 1 1 によって消滅させられる際に、次のスレッド “ n + 1 ” を生成する。以降は、上述した第 7 変形例と同様にして、順次、前のスレッド 1 3 が消滅する際に、次のスレッド 1 3 が生

成されて処理が実行される。

【0 1 2 4】

例えば、処理 A と処理 B の実行結果を処理 C が使う場合、スレッド 1 3 の構成例は図 2 6 に示すようになる。そして、このように構成したスレッド 1 3 を、この場合も、図 3 により前述したシステムに適用して設計の初期段階における論理装置のシミュレーションを行なう。

【0 1 2 5】

即ち、全てのスレッド 1 3 の処理が完了（全てのスレッド 1 3 が消滅）するまで、スレッドマネージャ 1 1 とリソースマネージャ 1 2 とが連携して、リソース 1 4 の要求、リソース 1 4 の割り当て及びリソース 1 4 の割り当て結果に応じたスレッド 1 3 の実行状態の制御を繰り返すことによって、論理装置の動作シミュレーションを実現する。

【0 1 2 6】

このように、本変形例では、スレッド 1 3 に並列性を導入して処理の並列化を図ることにより、上述した実施形態や第 7 変形例よりも複雑な「機能」をもった論理装置の動作シミュレーションを実現することができる。

【0 1 2 7】

（J）第 9 変形例の説明

次に、上述したリソースマネージャ 1 2 は、リソース 1 4 の種類に関係無く全てのリソース 1 4 を共通で管理していたが、例えば図 2 7 に模式的に示すように、複数種類のリソース 1 4 - 1 ~ 1 4 - n 毎にリソースマネージャ 1 2 - 1 ~ 1 2 - n を設けて、それぞれが 1 種類のリソース 1 4 - j（j = 1 ~ n）を独立して（ローカルに）管理して、要求 1 9 に応じたリソース 1 4 - j をローカルな「調停ルール」に従って要求元のスレッド 1 3 に割り当てるようにしてもよい。

【0 1 2 8】

この場合は計算機 1（CPU 4）を次のように動作させることが可能になる。即ち、外部入力（テストデータ 1 6 の入力；ステップ S 3 1）によりスレッドマネージャ 1 1 がスレッド 1 3 を生成する（ステップ S 3 2）。この場合も、スレッド 1 3 は独立して進行するが個々の実行順序はスレッドマネージャ 1 1 に管理

されている。そして、スレッド 1 3 は処理を進めるためにリソース 1 4 - j を確保する必要がある時は、そのリソース 1 4 - j の確保をスレッドマネージャ 1 1 経由で対応するリソースマネージャ 1 2 - j に要求する（ステップ S 3 3 ; リソース要求ステップ）。

【 0 1 2 9 】

リソースマネージャ 1 2 - j は単位時間内のリソース要求 1 9 をリソース 1 4 - j 毎にローカルに調停して、リソース 1 4 - j の割り当て結果をスレッドマネージャ 1 1 に回答 2 1 を発行する（ステップ S 3 4 ; リソース割り当てステップ）。スレッドマネージャ 1 1 はリソースマネージャ 1 2 - j からの回答 2 1 に基づいてスレッド 1 3 の実行状態を制御する（スレッド制御ステップ）。

【 0 1 3 0 】

そして、この場合も、全てのスレッド 1 3 の処理が完了するまで、図 3 により前述したごとく、スレッドマネージャ 1 1 とリソースマネージャ 1 2 - j とが連携して、リソース要求、リソース割り当て及びスレッド 1 3 の制御とを繰り返すことによって、実行結果を出力して（ステップ S 3 5）、論理装置の動作シミュレーションを実現する。

【 0 1 3 1 】

ここで、具体的に、リソース種類を A, B, C の 3 種類（以下、リソース A, B, C と表記する）とした場合、リソースマネージャ 1 2 - j が管理（参照）するリソース A は、C++ 記述スタイルで構成（記述）すると例えば図 2 8 に示すように構成される。即ち、前述したリソース要求キュー 2 0 とリソース回答キュー 2 2 とを定義するとともに、要求メソッド 3 1 a'、リソース A の解放を行なう解放メソッド 3 1 b'、リソース A の調停を行なう調停メソッド 3 1 c' をそれぞれ定義する。

【 0 1 3 2 】

ここで、要求メソッド 3 1 a' は、要求したリソース A とその要求 1 9 を出したスレッド 1 3 のスレッド ID とを要求メッセージとしてリソース要求キュー 2 0 に登録する操作を行なうための関数であり、解放メソッド 3 1 b' は、リソース A の解放を行ない、解放したリソース A の個数を 1 つ増やす操作を行なう関数

である。

【 0 1 3 3 】

また、調停メソッド 3 1 c' は、リソース A の割り当て調停を独立して（ローカルに）行なうための関数で、破線枠 3 1 1' 内の記述により、リソース要求キュー 2 0 が空でない限り、リソース要求キュー 2 0 から 1 つずつ要求 1 9 が取り出され、要求されたリソース A の個数がその時点で 0 以下であれば破線枠 3 1 2' 内の記述が実行され、要求されたリソース A の個数がその時点で 1 以上であれば破線枠 3 1 3' 内の記述が実行されるようになっている。

【 0 1 3 4 】

即ち、要求されたリソース A の個数がその時点で 0 以下であれば、新たに割り当てられるリソース A が無いので、リソース割り当て結果の回答 2 1 として “False” をリソース回答キュー 2 2 に格納し、要求されたリソース A の個数が 1 以上であれば、割り当て可能なので、そのリソース A の調停ルールに従って割り当てを行なって、その調停（割り当て）結果（“True”）を回答 2 1 としてリソース回答キュー 2 2 に格納するのである。なお、リソース B，リソース C についても上記（図 2 8）と同様に記述できる。つまり、この場合は、図 6 に示すものに比して記述量は増えるものの構成は簡単になる。

【 0 1 3 5 】

そして、このように構成したリソース A，B，C を図 3 に示すシステムに適用すれば、リソース種類毎のリソースマネージャ 1 2 - j によって、互いに依存関係の無いローカルな「調停ルール」に従ったリソース割り当てが行なわれて、リソース A，B，C の種類間に依存関係が存在しない論理装置のシミュレーションが可能となる。

【 0 1 3 6 】

（K）第 1 0 変形例の説明

なお、上述した第 9 変形例は、リソース種類に互いに依存関係が無い場合であるが、依存関係がある場合は、上述したローカルな調停では不十分である。例えば、「CPU」というリソースが、「論理演算装置（ALU；Arithmetic Logic Unit）」，「バス（BUS）」，「レジスタ（REGISTER）」などのリソースから構成され

、「ALU」というリソースが、さらに、「加算器 (ADDER)」, 「減算器 (SUBTRACTER)」, 「論理積回路 (AND)」, 「論理和回路 (OR)」などのリソースから構成されているような場合、「CPU」というリソースをスレッド13に割り当てるには、下位階層の「ALU」, 「BUS」, 「REGISTER」, 「ADDER」, 「AND」, 「OR」といったリソースも空いていなければならない。

【0137】

このように、リソースに14-jに依存関係がある場合には、例えば図29に模式的に示すように、その依存関係に応じてリソースマネージャ12-jを階層化する。

即ち、この図29の場合は、リソース14-1とリソース14-2との間には依存関係が無いため、それぞれリソースマネージャ12-1, 12-2がローカルにリソース14-1, 14-2を管理するが、リソース14-1, 14-2とリソース14-3との間にはそれぞれ依存関係が存在するため、上位階層のリソースマネージャ12-3が、下位階層のリソースマネージャ12-1, 12-2も管理することで下位階層のリソース14-1, 14-2の管理も行なえるようにするのである。

【0138】

同様にして、リソースマネージャ12-3やリソースマネージャ12-(n-1)は、互いに独立してリソース14-3や14-(n-1)の管理をローカルに行なうが、それぞれ上位階層のリソース14-nと依存関係があるため、上位階層のリソースマネージャ12-nが下位階層のリソースマネージャ12-3や12-(n-1)も管理する。

【0139】

このようにして、リソース14-jに依存関係があり前述したようなローカルなリソース調停で解決できない場合はリソース14-jの階層化を行なってグローバルなリソース調停を行なう。

なお、この場合、計算機1 (CPU4) は、次のように動作することになる。即ち、外部入力 (テストデータ16の入力; ステップS41) によりスレッドマネージャ11がスレッド13を生成する (ステップS42)。この場合も、スレ

ッド 1 3 は独立して進行するが個々の実行順序はスレッドマネージャ 1 1 に管理されている。

【 0 1 4 0 】

そして、スレッド 1 3 は処理を進めるためにリソース 1 4 - j を確保する必要がある時は、そのリソース 1 4 - j の確保を、スレッドマネージャ 1 1 経由でそのリソース種類に対応するリソースマネージャ 1 2 - j に要求する（ステップ S 4 3 ; リソース要求ステップ）。

【 0 1 4 1 】

リソースマネージャ 1 2 - j は、単位時間内の個々のリソース要求に対する割り当てをリソース 1 4 - j 毎にローカルに調停する。ローカルなリソース 1 4 - j の割り当て結果は上位階層のリソースマネージャ 1 2 - k ($k = j + 1$) に伝わり、上位階層のリソースマネージャ 1 2 - k は、他の関連する（依存関係のある）種類のリソース 1 2 - k の割り当て結果を考慮してその階層のリソース割り当てを決定する。

【 0 1 4 2 】

以降、同様にして、下位階層のリソース割り当て結果が、順次、上位階層に伝搬してゆき、最終的に、最上位階層のリソースマネージャ 1 2 - n が最後の割り当て結果をスレッドマネージャ 1 1 に回答する（ステップ S 4 4 ; リソース割り当てステップ）。

【 0 1 4 3 】

つまり、本変形例では、リソース割り当てステップ S 4 4 において、上位階層のリソースマネージャ 1 2 - k が、自身の管理するリソース 1 4 - k と下位階層のリソースマネージャ 1 2 - j の管理するリソース 1 4 - j との依存関係を考慮してリソース割り当てを行なうのである。

【 0 1 4 4 】

そして、スレッドマネージャ 1 1 は、このリソースマネージャ 1 2 - j からの回答 2 1 に基づいてスレッド 1 3 の実行状態を制御する（スレッド制御ステップ）。その後、全てのスレッド 1 3 の処理が完了するまで、スレッドマネージャ 1 1 とリソースマネージャ 1 2 - j とが連携して、上記のリソース要求、リソース

割り当て及びスレッド 1 3 の制御とを繰り返すことにより、実行結果（シミュレーション結果）を出力して（ステップ S 4 5）、論理装置のシミュレーションを実現する。

【0 1 4 5】

このように、本変形例では、リソース 1 4 - j の依存関係に応じてリソース 1 4 - j 及びリソースマネージャ 1 2 - j を階層化することで、上位階層のリソースマネージャ 1 2 - j が下位階層のリソースマネージャ 1 2 - j の管理するリソース 1 4 - j も考慮してリソース 1 4 - j の割り当てを行なうことができるので、リソース種類間のグローバルな調停を行なうことができ、より複雑なリソースの依存関係を持つ論理装置の動作シミュレーションを実現して、設計初期段階で機能や性能検証を行なうことができる。

【0 1 4 6】

なお、上述した第 7 ～ 第 1 0 変形例は、勿論、前述した第 1 ～ 第 6 変形例のいずれに適用してもよく、それぞれ、組み合わせによる利点ないし効果が得られる。

以上のように、本実施形態及びその各変形例の論理装置の動作シミュレーション方法によれば、論理装置の設計初期段階において機能レベルのシミュレーションを行なって、「機能」の確認、「機能」を実現するアーキテクチャの妥当性を設計の初期段階で確認して論理装置の機能検証と性能評価を行なうことができる。

【0 1 4 7】

また、設計の初期段階に「機能」を正確に実現できたか否かを確認することができるとともに、設計の安全性を確認することができる。その上、アーキテクチャに変更が必要となった場合でも、「機能」の記述の変更は殆ど行なわずに、リソース 1 4 （1 4 - j）、「調停ルール」、リソース要求などを変更するだけで、簡単に設計の再検証を行なえるので、従来のように「機能」の記述の変更による設計コスト増加を回避することができる。

【0 1 4 8】

（L）その他

上述した実施形態及びその各変形例では、シミュレーションプログラム 1 0 の記述の具体例として C ++ 記述スタイルを例にしたが、勿論、J a v a などの他のプログラミング言語に基づく記述スタイルを適用しても、上記と同様の作用効果が得られることはいうまでもない。

そして、本発明は、上述した実施形態及びその各変形例に限定されず、上記以外にも、本発明の趣旨を逸脱しない範囲で種々変形して実施することができる。

【 0 1 4 9 】

(M) 付記

〔付記 1〕 プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを、当該ハードウェアリソースを管理するリソースマネージャに要求するリソース要求ステップと

該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、

該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを有するとともに、

該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステップを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションすることを特徴とする、論理装置の動作シミュレーション方法。

【 0 1 5 0 】

〔付記 2〕 上記の一連の機能が、複数の逐次的なスレッドに表されていることを特徴とする、付記 1 記載の論理装置の動作シミュレーション方法。

〔付記 3〕 上記の一連の機能が、複数の逐次的または並行に実行されるスレッドに表されていることを特徴とする、付記 1 記載の論理装置の動作シミュレーション方法。

【 0 1 5 1 】

〔付記 4〕 該リソースマネージャが、該ハードウェアリソースの種類に対応

して複数設けられるとともに、

該リソース割り当てステップにおいて、上記の各リソースマネージャが、それぞれ、自身が管理するハードウェアリソースを予め規定されたローカルなルールに従って該スレッドに割り当てることを特徴とする、付記 1 ～ 3 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

【 0 1 5 2 】

〔付記 5〕 該リソースマネージャが、該ハードウェアリソースの種類に対応して複数設けられるとともに、各リソースマネージャが、該ハードウェアリソースの依存関係に応じて階層化され、

該リソース割り当てステップにおいて、該リソースマネージャが、自身の管理するハードウェアリソースと下位階層のリソースマネージャの管理するハードウェアリソースとの依存関係を考慮して該ハードウェアリソースの割り当てを行なうことを特徴とする、付記 1 ～ 3 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

【 0 1 5 3 】

〔付記 6〕 該リソースマネージャが、該リソース要求ステップによるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することを特徴とする、付記 1 ～ 5 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

〔付記 7〕 該リソースマネージャが、該リソース要求ステップによるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定することを特徴とする、付記 1 ～ 5 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

【 0 1 5 4 】

〔付記 8〕 該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該スレッドのボトルネックを検出することを特徴とする、付記 1 ～ 5 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

〔付記 9〕 該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該リソース要求のブロッキングを検出することを特徴とする、付記 1～5 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

【 0 1 5 5 】

〔付記 1 0〕 該スレッドに、該リソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておくことを特徴とする、付記 1～5 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

〔付記 1 1〕 該スレッドに、該機能の実行制限時間を与えておくことを特徴とする、付記 1～5 のいずれか 1 項に記載の論理装置の動作シミュレーション方法。

【 0 1 5 6 】

〔付記 1 2〕 付記 1～1 1 のいずれか 1 項に記載の動作シミュレーション方法によるシミュレーション結果と、該論理装置の動作予測値とを比較する比較ステップと、

該比較ステップでの比較結果を外部装置へ出力する出力ステップとを有することを特徴とする、論理装置の動作シミュレーション方法。

【 0 1 5 7 】

〔付記 1 3〕 プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャと、

該スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとをそなえとともに、

該スレッドマネージャが、

論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを該リソースマネージャに要求するリソース要求手段と、

該リソース要求手段による該要求に対する該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御手段とをそなえ、

且つ、

該リソースマネージャが、

該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当て手段をそなえ、

該スレッドの実行が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記のリソース要求及びスレッドの実行状態の制御を繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションすることを特徴とする、論理装置の動作シミュレーション装置。

【 0 1 5 8 】

〔付記 1 4〕 論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータを、

プログラムの実行単位であるスレッドの制御を行なうスレッドマネージャ及び該スレッドの実行に必要なハードウェアリソースを管理するリソースマネージャとして機能させるとともに、

該スレッドマネージャが、論理装置の設計仕様に応じて該論理装置の動作完了までに必要となる機能を表したスレッドの実行に必要なハードウェアリソースを該リソースマネージャに要求するリソース要求ステップと、該リソースマネージャが、該要求に応じたハードウェアリソースを予め規定されたルールに従って該スレッドに割り当てるリソース割り当てステップと、該スレッドマネージャが、該リソースマネージャによる割り当て結果に応じて該スレッドの実行状態を制御するスレッド制御ステップとを実行するとともに、該スレッドの処理が完了するまで該スレッドマネージャと該リソースマネージャとが連携して上記の各ステップを繰り返し実行することにより、該論理装置の動作完了までの動作をシミュレーションするためのプログラムが記録されていることを特徴とする、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 5 9 】

〔付記 1 5〕 上記の一連の機能が、複数の逐次的なスレッドに表されている

ことを特徴とする、付記 1 4 記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

〔付記 1 6〕 上記の一連の機能が、複数の逐次的もしくは並行に実行されるスレッドに表されていることを特徴とする、付記 1 4 記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 0 】

〔付記 1 7〕 該リソースマネージャが、該ハードウェアリソースの種類に対応して複数設けられるとともに、

該リソース割り当てステップにおいて、上記の各リソースマネージャが、それぞれ、自身が管理するハードウェアリソースを予め規定されたローカルなルールに従って該スレッドに割り当てることを特徴とする、付記 1 4 ～ 1 6 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 1 】

〔付記 1 8〕 該リソースマネージャが、該ハードウェアリソースの種類に対応して複数設けられるとともに、各リソースマネージャが、該ハードウェアリソースの依存関係に応じて階層化され、

該リソース割り当てステップにおいて、該リソースマネージャが、自身の管理するハードウェアリソースと下位階層のリソースマネージャの管理するハードウェアリソースとの依存関係を考慮して該ハードウェアリソースの割り当てを行なうことを特徴とする、付記 1 4 ～ 1 6 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 2 】

〔付記 1 9〕 該リソースマネージャが、該リソース要求ステップによるリソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することを特徴とする、付記 1 4 ～ 1 8 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 3 】

〔付記 2 0〕 該リソースマネージャが、該リソース要求ステップによるリソース要求によって割り当てられたハードウェアリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのハードウェアリソースに対するリード／ライト動作の競合状態を判定することを特徴とする、付記 1 4 ～ 1 8 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 4 】

〔付記 2 1〕 該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該スレッドのボトルネックを検出することを特徴とする、付記 1 4 ～ 1 8 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 5 】

〔付記 2 2〕 該リソースマネージャが、該ハードウェアリソースに対するリソース要求数を監視し、その監視結果に基づいて該リソース要求のブロッキングを検出することを特徴とする、付記 1 4 ～ 1 8 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 6 】

〔付記 2 3〕 該スレッドに、該リソースマネージャによって割り当てられたハードウェアリソースを占有する時間についての予算を与えておくことを特徴とする、付記 1 4 ～ 1 8 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

〔付記 2 4〕 該スレッドに、該機能の実行制限時間を与えておくことを特徴とする、付記 1 4 ～ 1 8 のいずれか 1 項に記載の論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 7 】

〔付記 2 5〕 論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体であって、

該コンピュータに、

付記 1 ～ 1 1 のいずれか 1 項に記載の動作シミュレーション方法によるシミュレーション結果と該論理装置の動作予測値とを比較する比較ステップと、該比較ステップでの比較結果を外部装置へ出力する出力ステップとを実行させるための動作シミュレーションプログラムが記録されたことを特徴とする、論理装置の動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体。

【 0 1 6 8 】

【発明の効果】

以上詳述したように、本発明によれば、論理装置に実装すべき「機能」が必要とするハードウェアリソース（以下、単に「リソース」という）に関する記述はスレッドには行なわず、そのスレッドが実行されるときに、その都度、必要なリソースをリソースマネージャによって動的に割り当てながら、スレッドの実行が完了するまでスレッドマネージャとリソースマネージャとが連携してスレッドの実行状態の制御を繰り返し実行することにより、論理装置の動作完了までの動作をシミュレーションするので、次のような利点が得られる。

【 0 1 6 9 】

①論理装置の動作予測値との比較により、「機能」の確認、「機能」を実現するアーキテクチャの妥当性を設計の初期段階で確認して論理装置の機能検証と性能評価を行なうことができる。

②論理装置の動作予測値との比較により、設計の初期段階に「機能」を正確に実現できたか否かを確認することができる。

【 0 1 7 0 】

③アーキテクチャに変更が必要になった場合でも、「機能」の記述変更は殆ど行なわずに、リソースやリソースの割り当てルール、スレッドのスケジューリング、リソース要求などを、アーキテクチャの変更に応じて変更するだけで、簡単に設計の再検証を行なうことが可能になり、「機能」の記述の変更による設計コスト増加を回避することができる。

【 0 1 7 1 】

④「機能」に適したアーキテクチャの探索や、「機能」をアーキテクチャへマッピングする際のアーキテクチャの性能評価、設計初期段階におけるタイミングの検証などが行なえる。

なお、論理装置の動作完了までに必要な一連の「機能」は、複数の逐次的なスレッド群に表されていてもよいし、複数の逐次的または並行に実行されるスレッド群に表されていてもよい。前者の場合は、論理装置の「機能」の依存関係をより明確にした上での動作シミュレーションが可能になり、後者の場合はより複雑な「機能」構成をもった論理装置の動作シミュレーションが可能になる。

【 0 1 7 2 】

また、上記のリソースマネージャは、リソースの種類に対応して複数設けられていてもよく、例えば、それぞれが管理するリソースを予め規定されたローカルなルールに従ってスレッドに割り当てたり、各リソースマネージャをリソースの依存関係に応じて階層化して、それぞれの管理するリソースと下位階層のリソースマネージャの管理するリソースとの依存関係を考慮してリソースの割り当てを行なったりすることも可能である。

【 0 1 7 3 】

前者の場合は簡単な記述（構成）でリソースに依存関係のない論理装置のシミュレーションが可能となり、後者の場合はリソースの依存関係をも考慮したグローバルなリソースの割り当て調停が可能となる。

さらに、上記のリソースマネージャは、リソース要求を監視し、その監視結果に基づいて複数スレッド間のリソース要求のデッドロック状態を判定することもでき、この場合は、設計の初期段階においてデッドロック状態の発生可能性を検出することが可能となり、論理装置の機能検証を行なうことが可能となる。

【 0 1 7 4 】

また、上記のリソースマネージャは、リソース要求によって割り当てられたリソースに対するリード／ライト要求を監視し、その監視結果に基づいて複数スレッドのリソースに対するリード／ライト動作の競合状態を判定することもでき、この場合は、複数のスレッドがリソースに対してシーケンシャルに書き込み動作もしくは読み出し動作を行なう時に、その順序が正しいか否かを判定して、設計

した論理装置が正しく動作するか否かを検証することができる。

【 0 1 7 5 】

さらに、上記のリソースマネージャは、リソースに対するリソース要求数を監視し、その監視結果に基づいてスレッドのボトルネックを検出することもでき、この場合は、アクセス頻度が最も多いリソースは論理装置のボトルネックになっている可能性があるとして推測することができ、論理装置の設計初期段階で論理装置の性能検証（ボトルネックの有無の検証）を実現することができる。

【 0 1 7 6 】

また、上記のリソースマネージャは、リソースに対するリソース要求数を監視し、その監視結果に基づいてリソース要求のブロッキングを検出することもできる。この場合は、リソース要求のブロッキングの発生可能性を検出することで、論理装置の性能を評価することができる。

【 0 1 7 7 】

さらに、上記のスレッドには、リソースマネージャによって割り当てられたリソースを占有する時間についての予算を与えておいてもよい。このようにすれば、各種「機能」の組み合わせで構成される論理装置が、設計仕様により要求される処理時間に関する性能要件を満たしているか否かを、設計の初期段階にて確認することができる。

【 0 1 7 8 】

また、上記のスレッドには、上記機能の実行制限時間を与えておいてもよく、このようすれば、設計の初期段階においてリアルタイム性に関する性能要件を論理装置が満足しているかを検証することが可能となる。

さらに、上述した動作シミュレーション方法は、コンピュータを上述のごとく動作させる動作シミュレーションプログラムを記録したコンピュータ読み取り可能な記録媒体によって提供されてもよく、このようすれば、コンピュータに上記プログラムを読み取らせれば（インストールすれば）、そのコンピュータを上述したような論理装置の動作シミュレーション方法を実行する装置（論理装置の動作シミュレーション装置）として機能させることができるので、その普及に大きく寄与する。

【図面の簡単な説明】

【図 1】

本発明の第 1 実施形態に係るパーソナルコンピュータなどの計算機（情報処理装置）の構成を示すブロック図である。

【図 2】

本発明の第 1 実施形態に係る動作シミュレーションプログラムを説明するためのオブジェクトモデル図である。

【図 3】

図 2 に示す動作シミュレーションプログラムに基づく計算機の動作（論理装置の動作シミュレーション方法）を説明するための図である。

【図 4】

図 1 に示す計算機の要部に着目した構成を示すブロック図である。

【図 5】

第 1 実施形態に係るスレッドの記述例を示す図である。

【図 6】

第 1 実施形態に係るリソースマネージャの記述例を示す図である。

【図 7】

第 1 実施形態に係る論理装置（QoS システム）に必要な「機能」を説明するためのブロック図である。

【図 8】

第 1 実施形態に係る論理装置（QoS システム）の設計手順（動作シミュレーション方法）を説明するためのフローチャートである。

【図 9】

第 1 実施形態の第 1 変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図 10】

第 1 変形例に係るスレッドの記述例を示す図である。

【図 11】

（A）及び（B）はそれぞれ第 1 変形例に係るスレッドのデッドロックの検出

を説明するためのリソース割付けグラフである。

【図 1 2】

第 1 実施形態の第 2 変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図 1 3】

第 2 変形例に係るリソース要求の記述例を示す図である。

【図 1 4】

第 2 変形例に係るリソースの記述例を示す図である。

【図 1 5】

第 1 実施形態の第 3 変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図 1 6】

第 3 変形例に係るリソースの記述例を示す図である。

【図 1 7】

第 1 実施形態の第 4 変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図 1 8】

第 4 変形例に係るリソースの記述例を示す図である。

【図 1 9】

第 1 実施形態の第 5 変形例に係る論理装置の設計手順（動作シミュレーション方法）を説明するためのフローチャートである。

【図 2 0】

第 5 変形例に係るスレッドの記述例を示す図である。

【図 2 1】

第 1 実施形態の第 6 変形例に係る論理装置の設計手順（動作シミュレーション方法）を説明するためのフローチャートである。

【図 2 2】

第 6 変形例に係るスレッドの記述例を示す図である。

【図 2 3】

第 1 実施形態の第 7 変形例に係るスレッドの生成方法を説明するための模式図である。

【図 2 4】

第 7 変形例に係るスレッドの記述例を示す図である。

【図 2 5】

第 1 実施形態の第 8 変形例に係るスレッドの生成方法を説明するための模式図である。

【図 2 6】

第 8 変形例に係るスレッドの記述例を示す図である。

【図 2 7】

第 1 実施形態の第 9 変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図 2 8】

第 9 変形例に係るリソースの記述例を示す図である。

【図 2 9】

第 1 実施形態の第 1 0 変形例に係る計算機の要部に着目した構成を示すブロック図である。

【図 3 0】

(A) 及び (B) はいずれも従来の論理装置の設計手順を説明するためのフローチャートである。

【図 3 1】

従来の論理装置の設計手順による課題を説明するための図である。

【符号の説明】

- 1 計算機 (情報処理装置 ; 論理装置の動作シミュレーション装置)
- 2 計算機本体
- 3 ディスプレイ (表示装置 ; 外部装置)
- 4 C P U
- 5 主記憶部 (メモリ)
- 6 二次記憶装置 (ハードディスク)

- 7 フロッピーディスク (FD) ドライブ
- 8 内部バス
- 9 フロッピーディスク (コンピュータ読み取り可能な記録媒体)
- 10, 51 論理装置の動作シミュレーションプログラム
- 11 スレッドマネージャ
- 12, 12-1~12-n リソースマネージャ
- 13 スレッド
- 14, 14-1~14-n リソース (ハードウェア資源)
- 15 テストベンチ
- 16 テストデータ
- 17 入力キュー
- 18 実行待ちキュー
- 19 要求
- 20 リソース要求キュー
- 21 回答
- 22 リソース回答キュー
- 31a, 31a' 要求メソッド
- 31b, 31b' 解放メソッド
- 31c, 31c' 調停メソッド
- 41 パケット
- 42 クラス分け機能
- 43 キューイング機能
- 44 デキュー機能
- 45 キュー
- 110 スレッド生成手段
- 111 リソース要求手段
- 112 スレッド制御手段
- 121 リソース割り当て手段
- 122 デッドロック検出機構

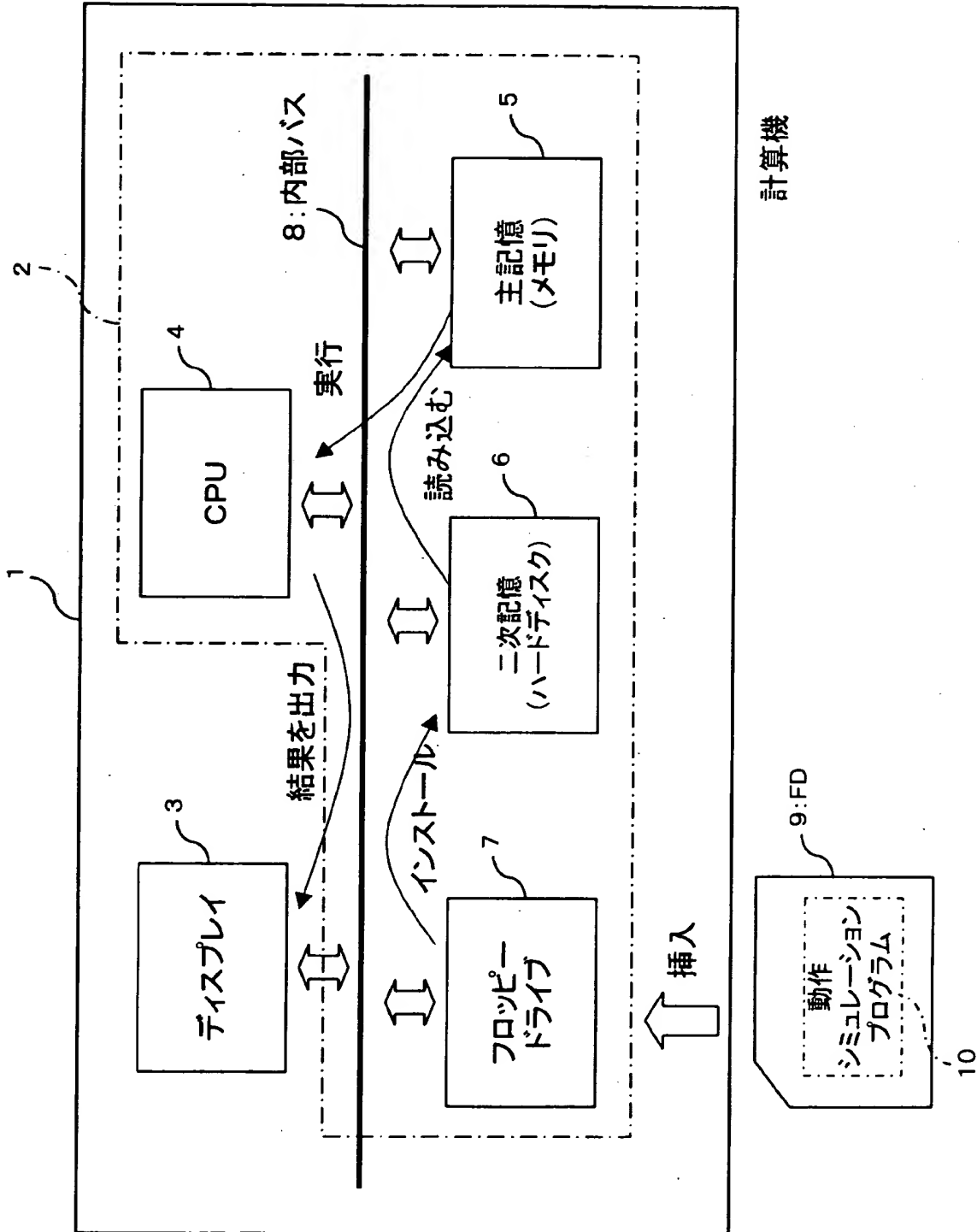
1 2 3 リード／ライト検出機構

1 2 4 ボトルネック検出機構

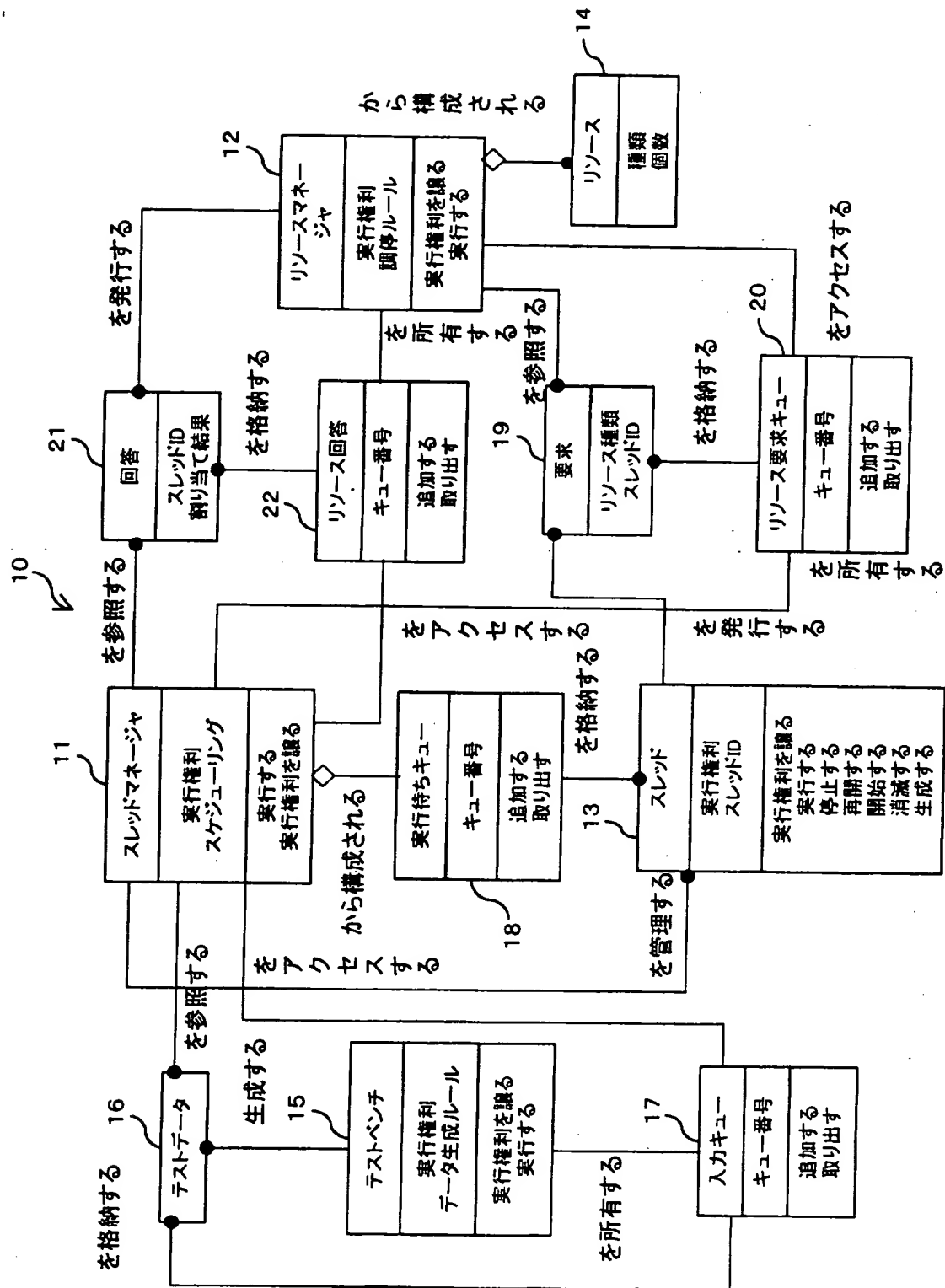
1 2 5 ブロッキング検出機構

【書類名】 図面

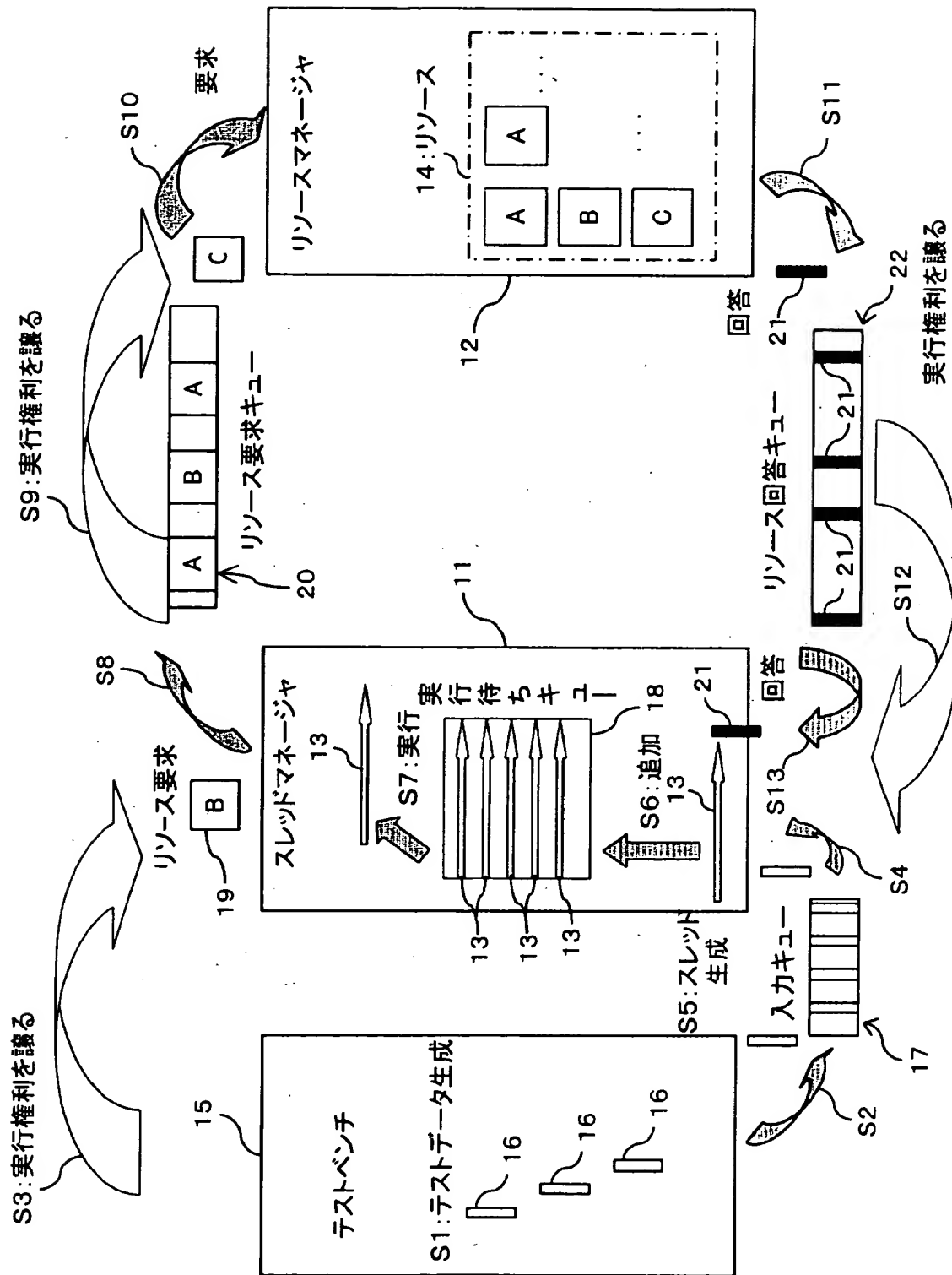
【図 1】



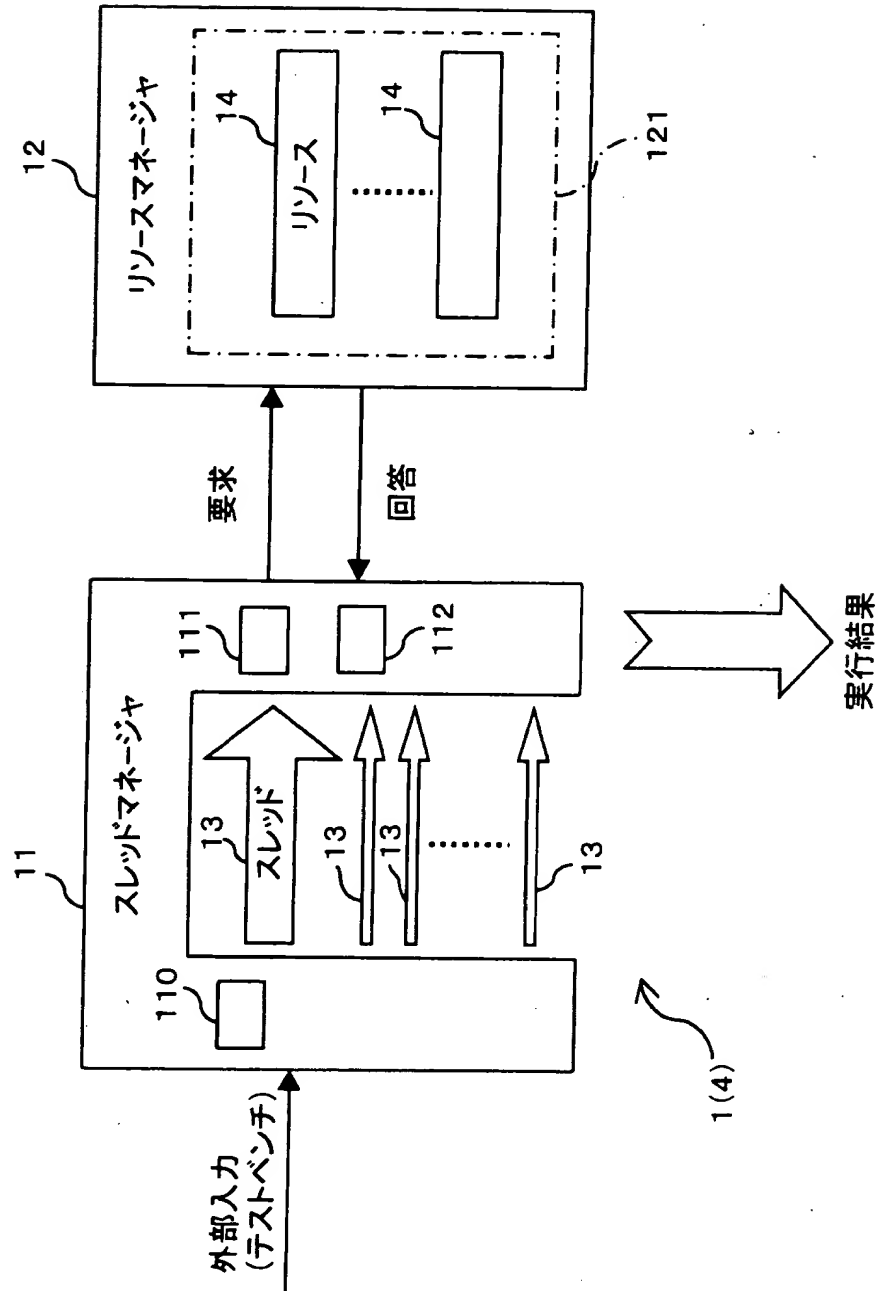
【圖 2】



【図 3】



【図 4】



【図 5】

```
class スレッド{  
    ...  
    void 実行(){  
        処理A();  
        処理B();  
        処理C();  
    }  
}
```

【図 6】

```

class リソースマネージャ{
    リソース R1;
    リソース R2;
    queue リソース要求キュー ← 20
    queue リソース回答キュー ← 22
public:
    void 要求 (リソース R){
        要求.スレッド ID = 現在のスレッド ID;
        要求.リソースの種類 = R;
        リソース要求キューに要求を追加
    }
    void 解放 (リソース R){
        R.個数 ++;
    }
    bool 調停
    while (リソース要求キュー空ではない){
        1つ要求を取り出す;
        if (要求.リソースの種類 == R1){
            if (R1.個数 <= 0){
                回答.スレッド ID = 要求.スレッド ID;
                回答.割り当て結果 = false;
                リソース回答キューに回答を追加
                continue;
            }
            R1 の調停ルール;
            回答.スレッド ID = 要求.スレッド ID;
            回答.割り当て結果 = R1 の調停結果;
            リソース回答キューに回答を追加する
            R1.個数 --;
        }
        else if (要求.リソースの種類 == R2){
            if (R2.個数 <= 0){
                回答.スレッド ID = 要求.スレッド ID;
                回答.割り当て結果 = false;
                リソース回答キューに回答を追加
                continue;
            }
            R2 の調停ルール;
            回答.スレッド ID = 要求.スレッド ID;
            回答.割り当て結果 = R2 の調停結果;
            リソース回答キューに回答を追加する
            R2.個数 --;
        }
    }
}

```

31a

31b

311

312

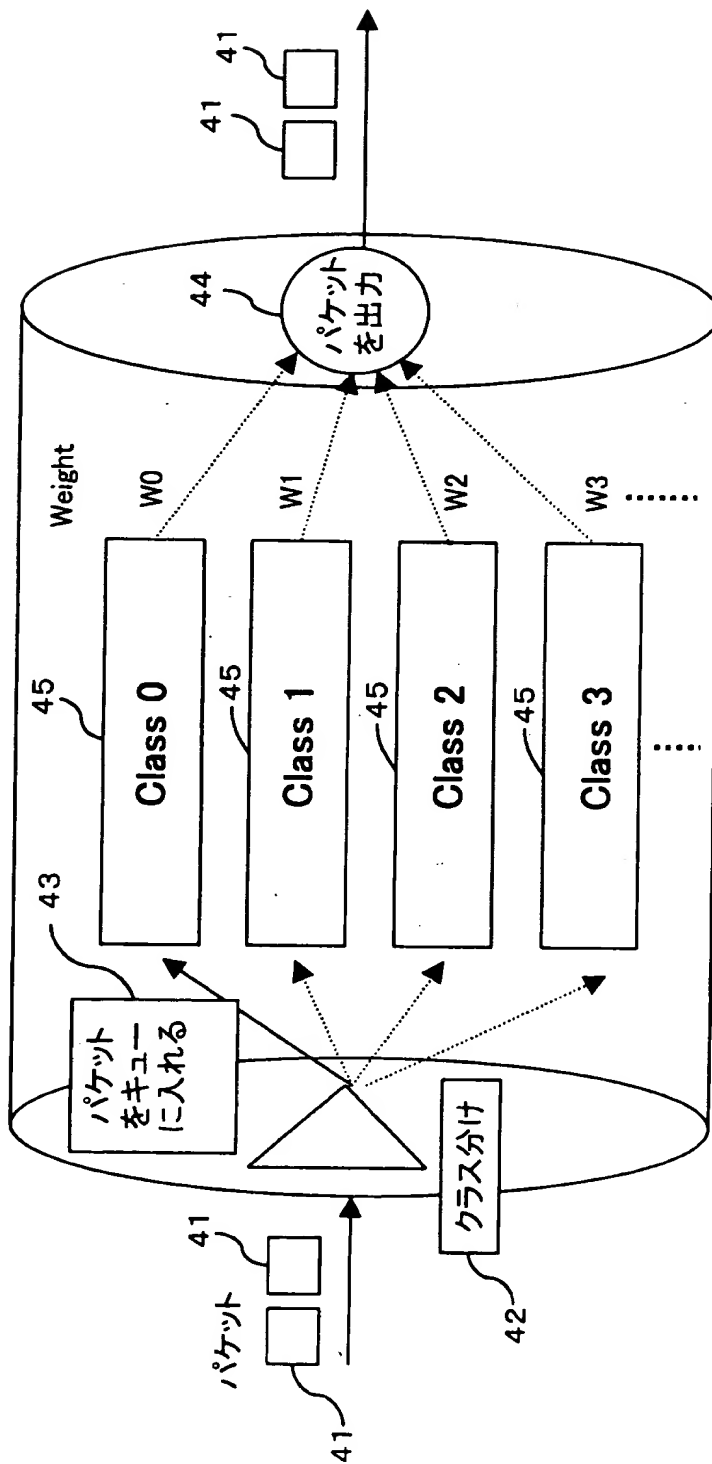
31c

313

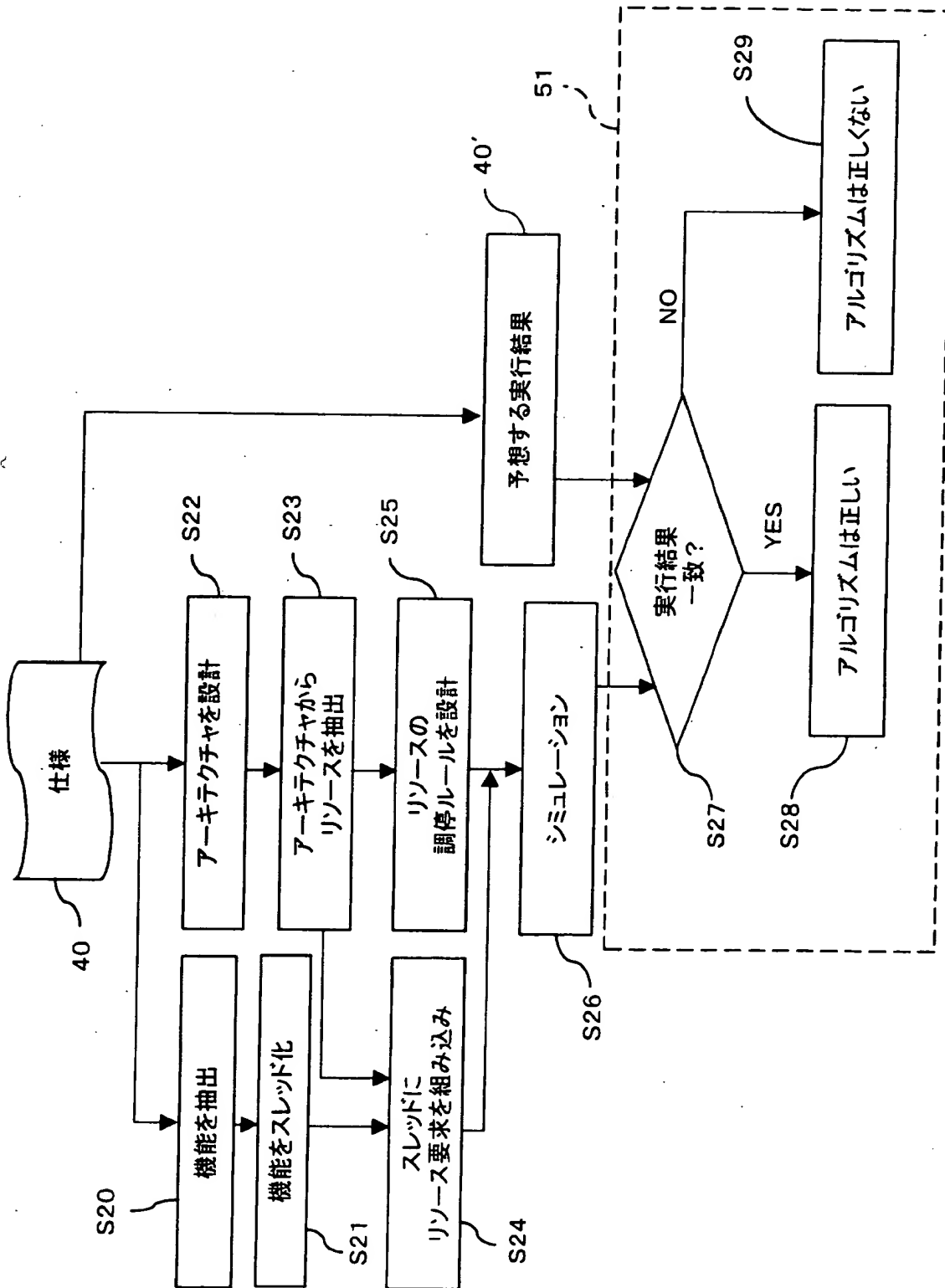
314

315

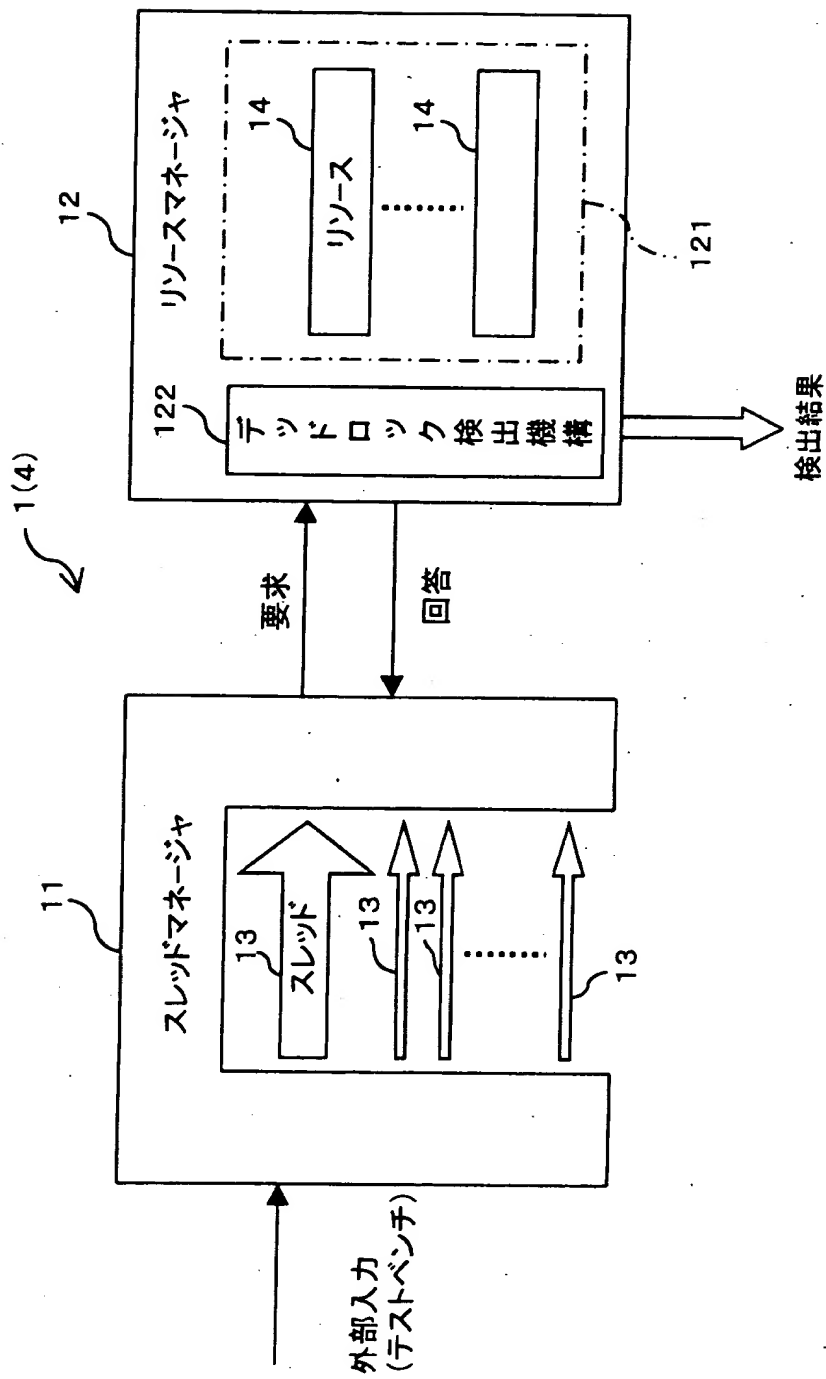
【図 7】



【図 8】



【図 9】



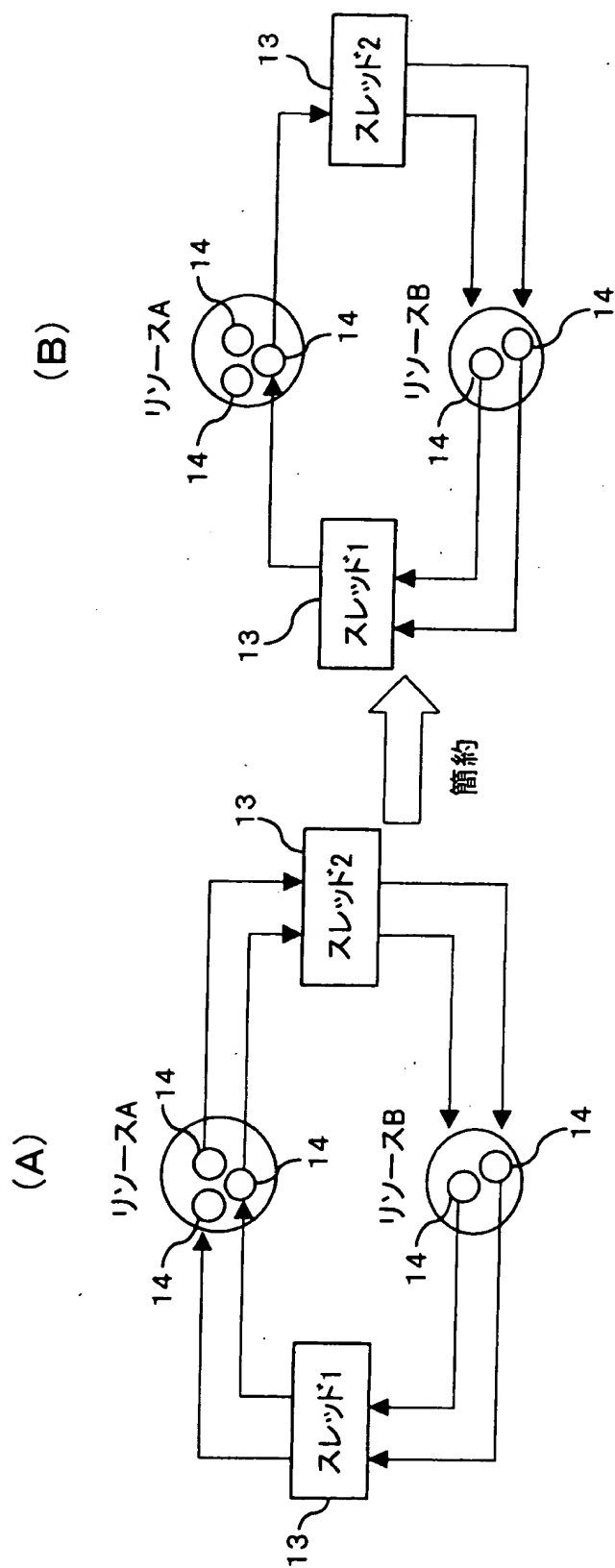
【図 1 0】

```

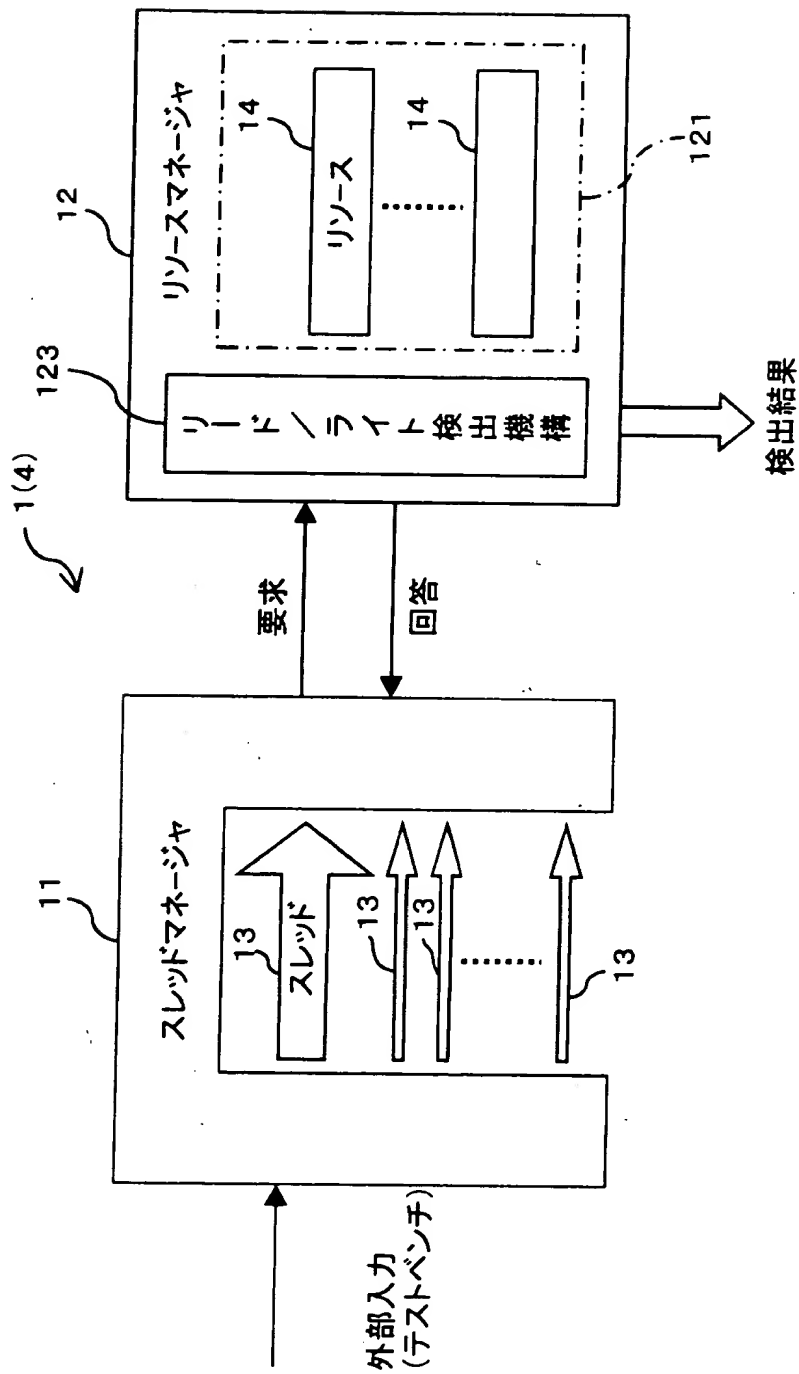
class スレッド{
    ...
    void 実行(){
        リソース A.要求(2);
        処理;
        ...
        リソース B.要求(2);
        リソース A.解放(2);
        処理;
        ...
        リソース A.要求(2);
        リソース B.解放(2);
        処理;
        ...
        リソース A.解放(2);
        ...
    }
}

```

【図 11】



【図 12】



【図 1 3】

```

class 要求{
    unsigned int スレッド ID;
    int 要求個数;
    int リード／ライトフラグ;
}

```

【図 1 4】

```

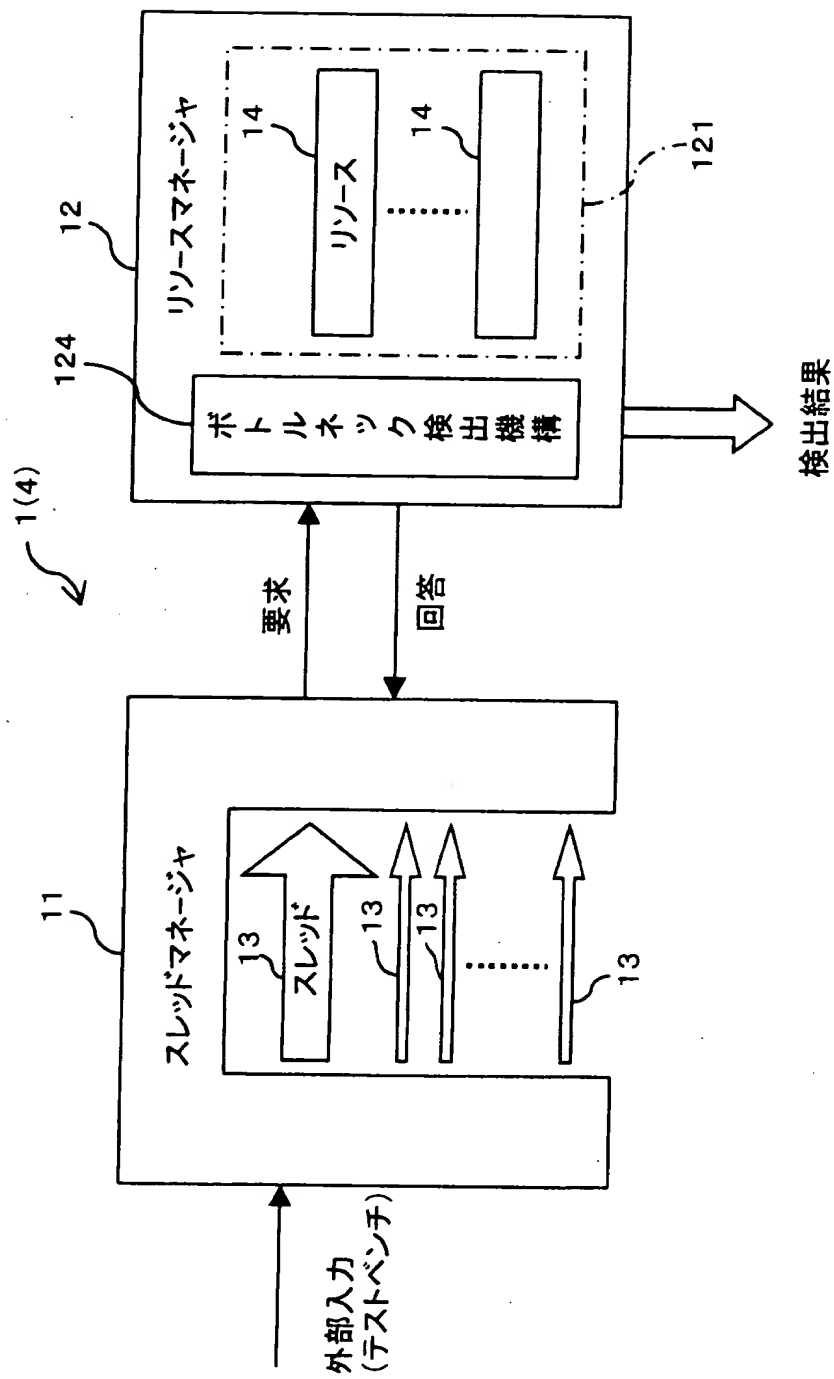
class リソース A: public リソース{
    int CurrentFlag = 0;
    ...
    void 要求(int n, bool ReadWriteFlag){
        要求.スレッド ID = 現在のスレッド ID;
        要求.リソース数 = n;
        要求.リード／ライトフラグ = ReadWriteFlag;
        リソース要求キューに要求を追加する。
    }
    void 解放(int n){
        ...
        CurrentFlag = 0;
    }
    ...
    bool 調停(){
        ...
        while (リソース要求キュー空ではない){
            1つ要求を取り出す;
            if (CurrentFlag != 0 && 要求.リード／ライトフラグ
                != CurrentFlag){
                error("リード／ライトエラーの発生可能性がある!");
            }
            CurrentFlag == 要求.リード／ライトフラグ;
        }
        ...
    }
    ...
}

```

316

317

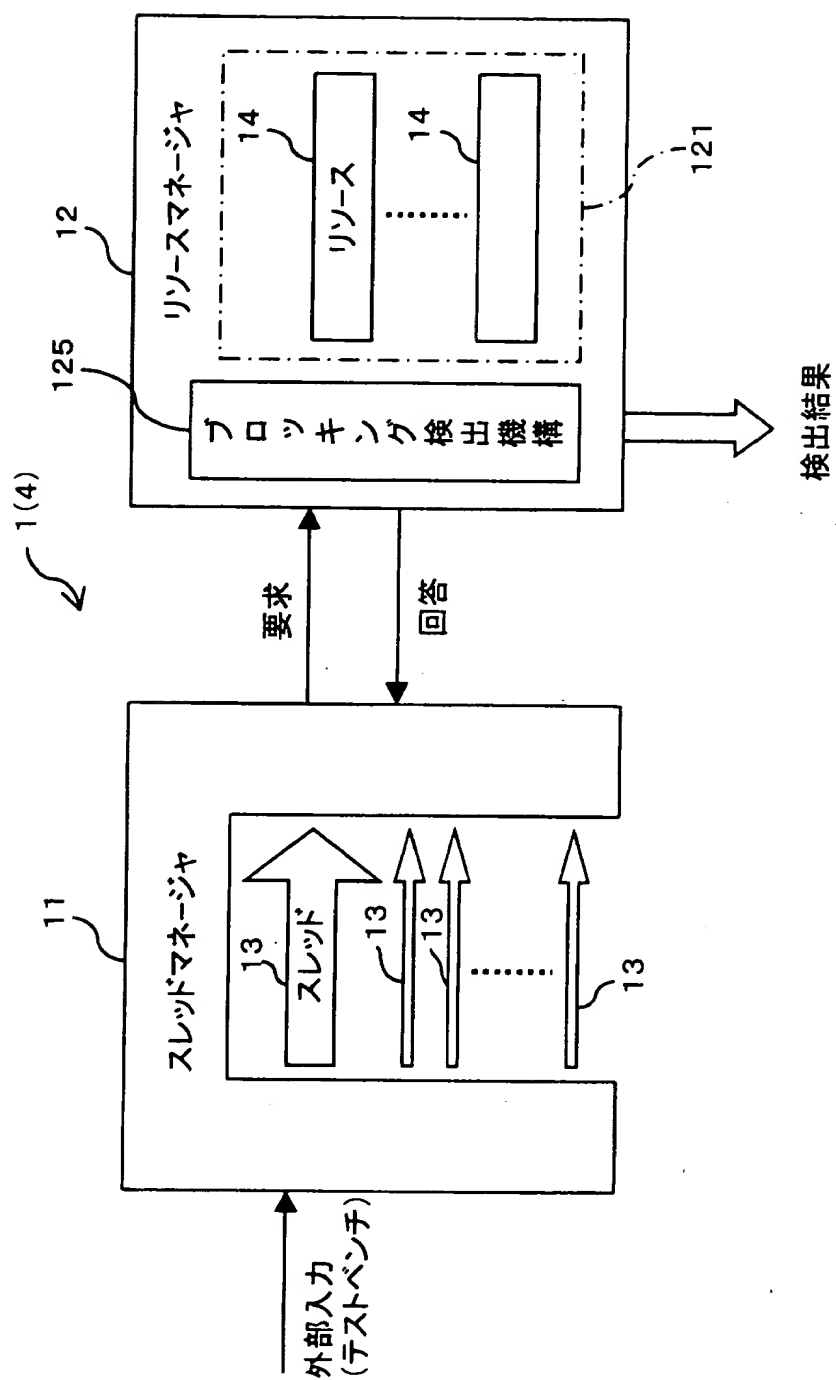
【図15】



【図 1 6】

```
class リソース {  
    int アクセス数 = 0 ;  
    ...  
    void 要求(int n){  
        アクセス数++;  
    }  
    ...  
    int 要求合計(){  
        return アクセス数;  
    }  
}
```

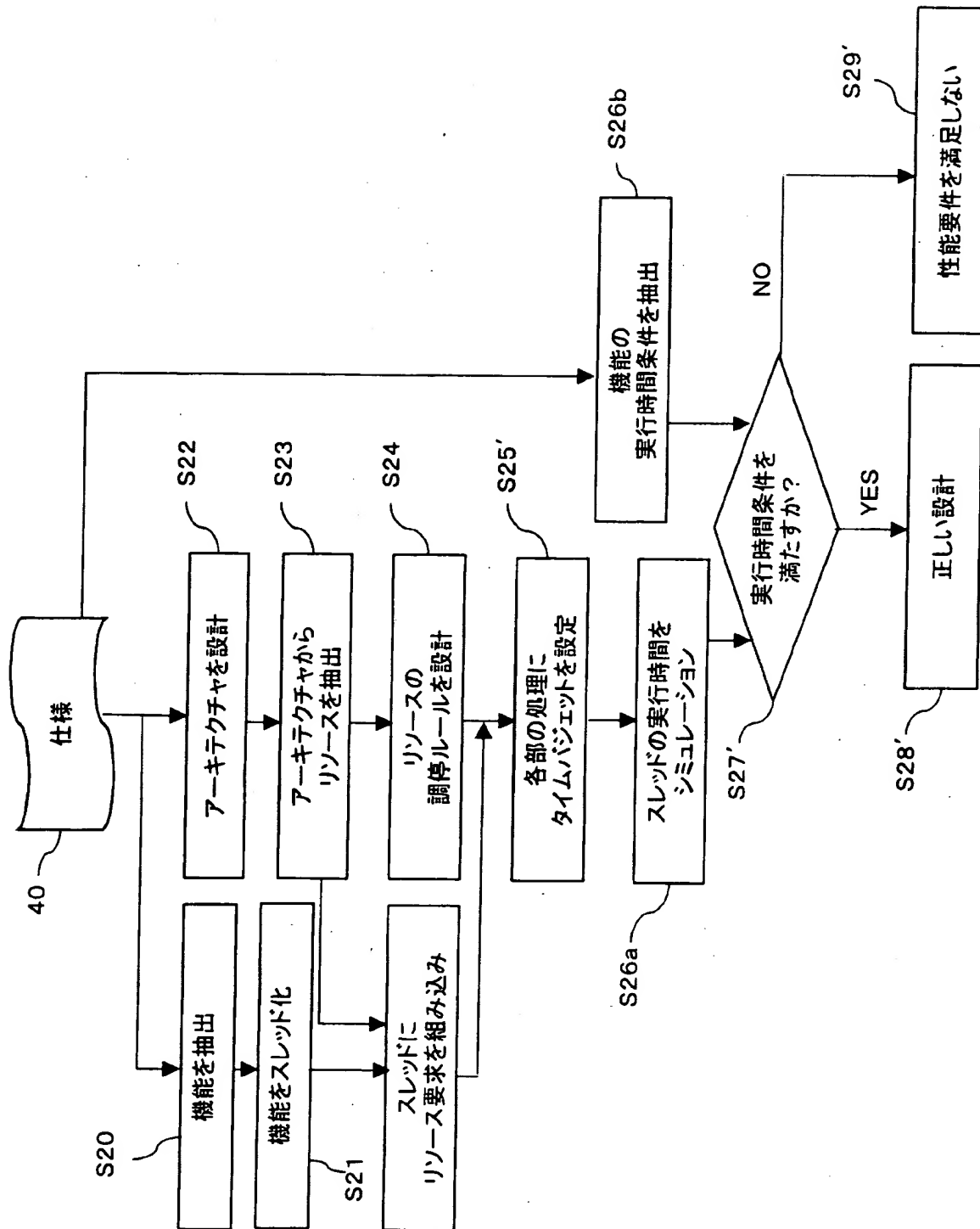
【図 17】



【図 1 8】

```
class リソース A{  
    ...  
    bool 調停(){  
        while (リソース要求キュー空ではない){  
            一つ要求を取り出す;  
        }  
        if (個数 < 0){  
            error("リソース A の要求はブロッキングする必要がある");  
        }  
        ...  
    }  
}
```

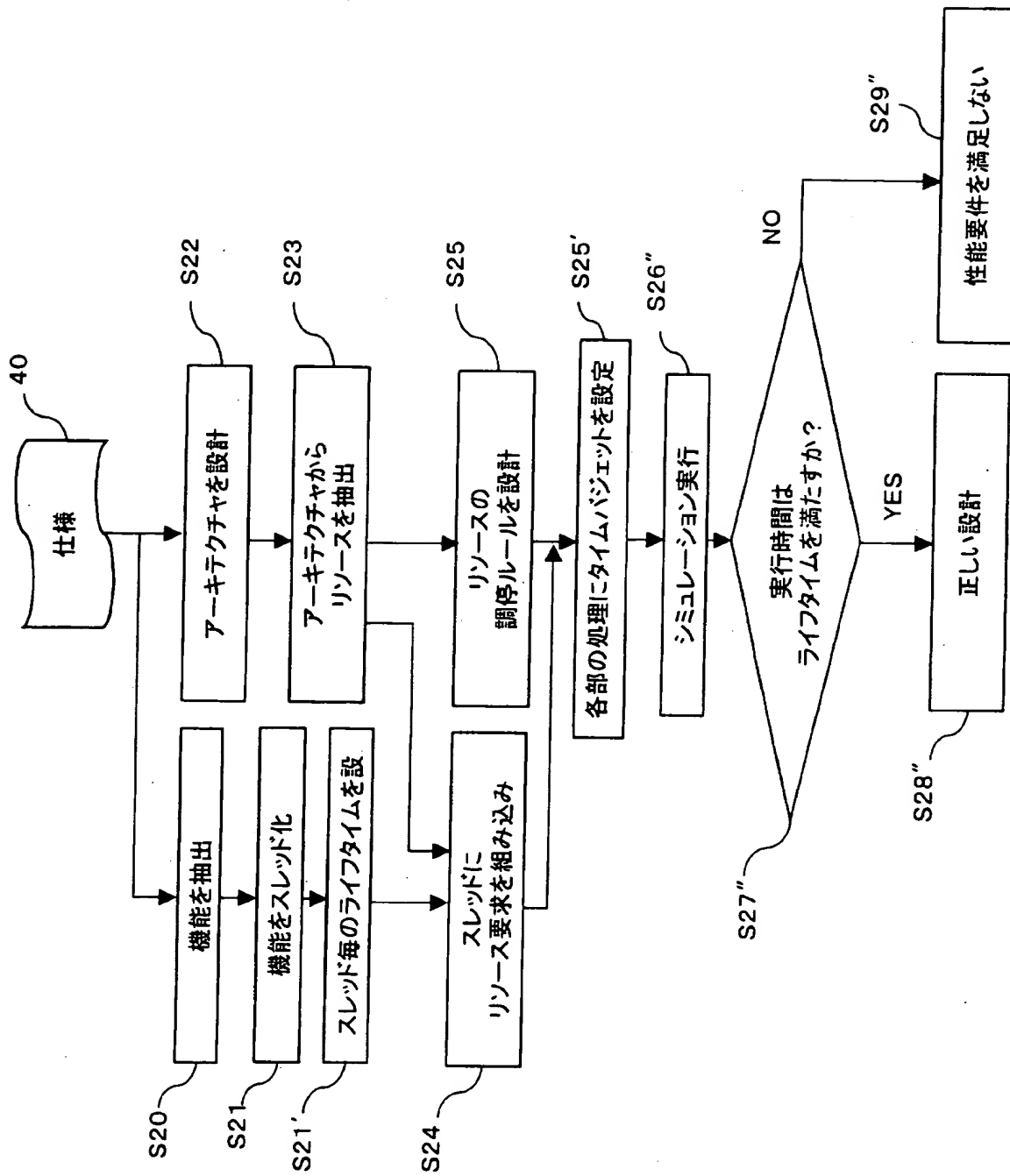
【図 19】



【図 2 0】

```
class スレッド{  
    ...  
    void 実行(){  
        リソースA.要求(1);  
        処理 1;  
        delay(処理 1 のタイムバジェット);  
        リソースA.解放(1);  
        リソースB.要求(1);  
        処理 2;  
        delay(処理 2 のタイムバジェット);  
        リソースB.解放(1);  
        ...  
    }  
}
```

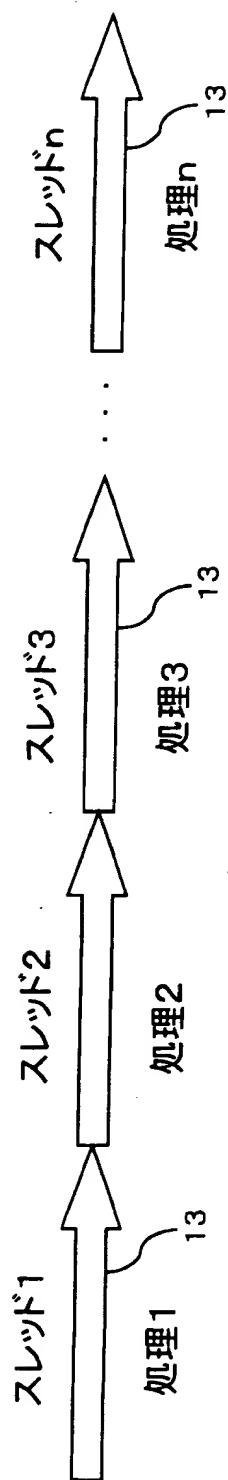
【図 21】



【図 2 2】

```
class スレッド {  
    long ライフタイム;  
    long 遅延;  
    void 実行(){  
        ...  
    }  
    bool JudgeLifeTime(){  
        if (ライフタイム < 遅延)  
            return false;  
        return true;  
    }  
}
```

【図 2 3】



【図 2 4】

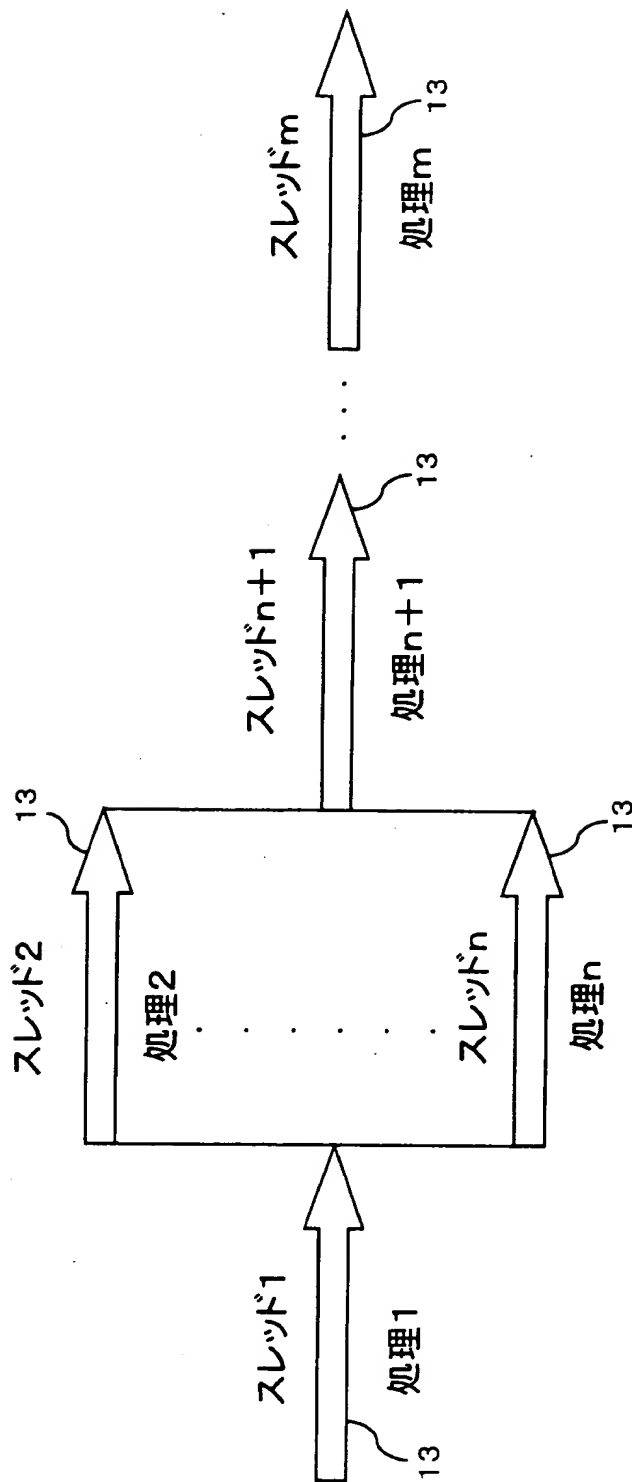
```
class スレッド{
    ...
    void 実行(){
        スレッド 1.生成();
        スレッド 1.終了を待つ();
        スレッド 2.生成();
        スレッド 2.終了を待つ();
        スレッド 3.生成();
        スレッド 3.終了を待つ();
    }
}

class スレッド 1: public スレッド{
    ...
    void 実行(){
        処理 1();
    }
    ...
}

class スレッド 2: public スレッド{
    ...
    void 実行(){
        処理 2();
    }
    ...
}

スレッド 3: public スレッド{
    ...
    void 実行(){
        処理 3();
    }
    ...
}
```

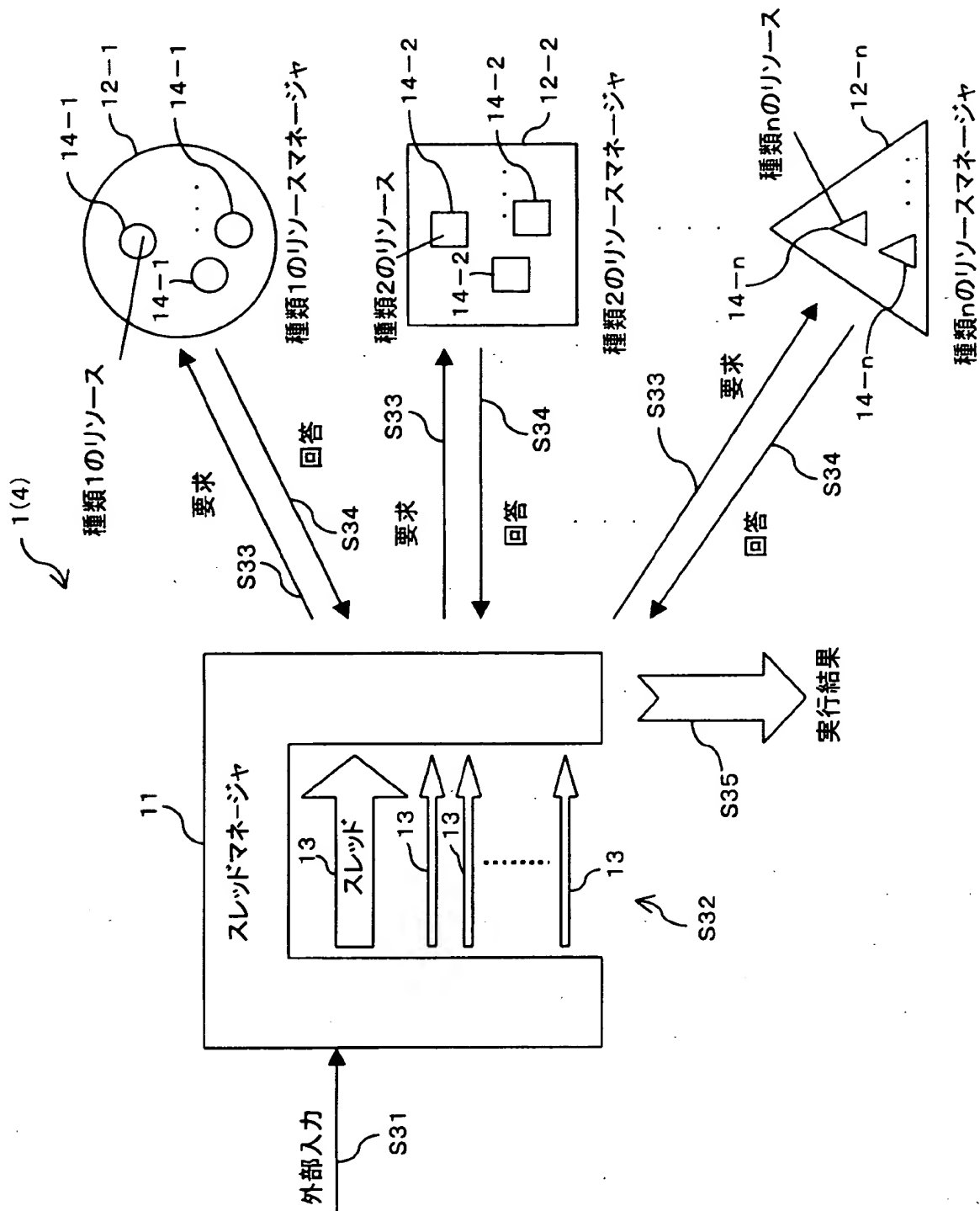
【図 25】



【図 2 6】

```
class スレッド{
    ...
    void 実行(){
        スレッド A.生成();
        スレッド B.生成();
        スレッド A.終了を待つ();
        スレッド B.終了を待つ();
        スレッド C.生成();
        スレッド C.終了を待つ();
    }
}
```

【図 27】



【図 28】

```

class リソースA: public リソースマネージャ{
    int 個数;
    queue リソース要求キュー; ← 20
    queue リソース回答キュー; ← 22
public:
    リソースA(int cnt): 個数(cnt){}
    void 要求(int n){
        要求.スレッドID = 現在のスレッドID;
        要求.リソース数 = n;
        リソース要求キューに要求を追加
    }
    void 解放(int n){
        個数 += n;
        if(個数 > cnt) 個数 = cnt;
    }
    bool 調停(){
        while (リソース要求キュー空ではない){
            一つ要求を取り出す;
            if(個数 <= 0){
                回答.スレッドID = 要求.スレッドID;
                回答.割り当て結果 = false;
                リソース回答キューに回答を追加
                continue;
            }
            結果 = リソースAの調停ルールに従って調停;
            回答.スレッドID = 要求.スレッドID;
            回答.割り当て結果 = 結果;
            リソース回答キューに回答を追加
            個数 --;
        }
    }
}

```

31a'

31b'

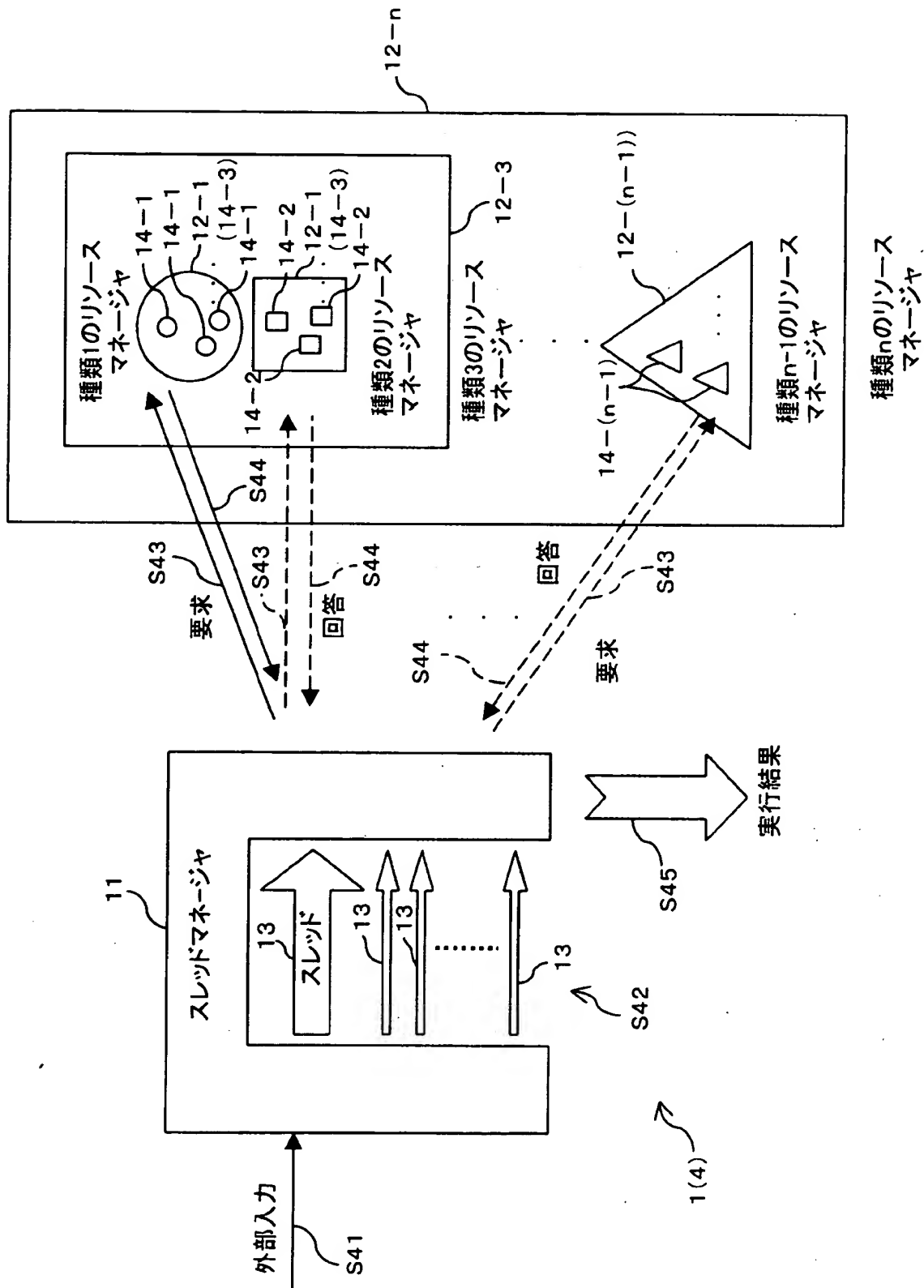
311'

312'

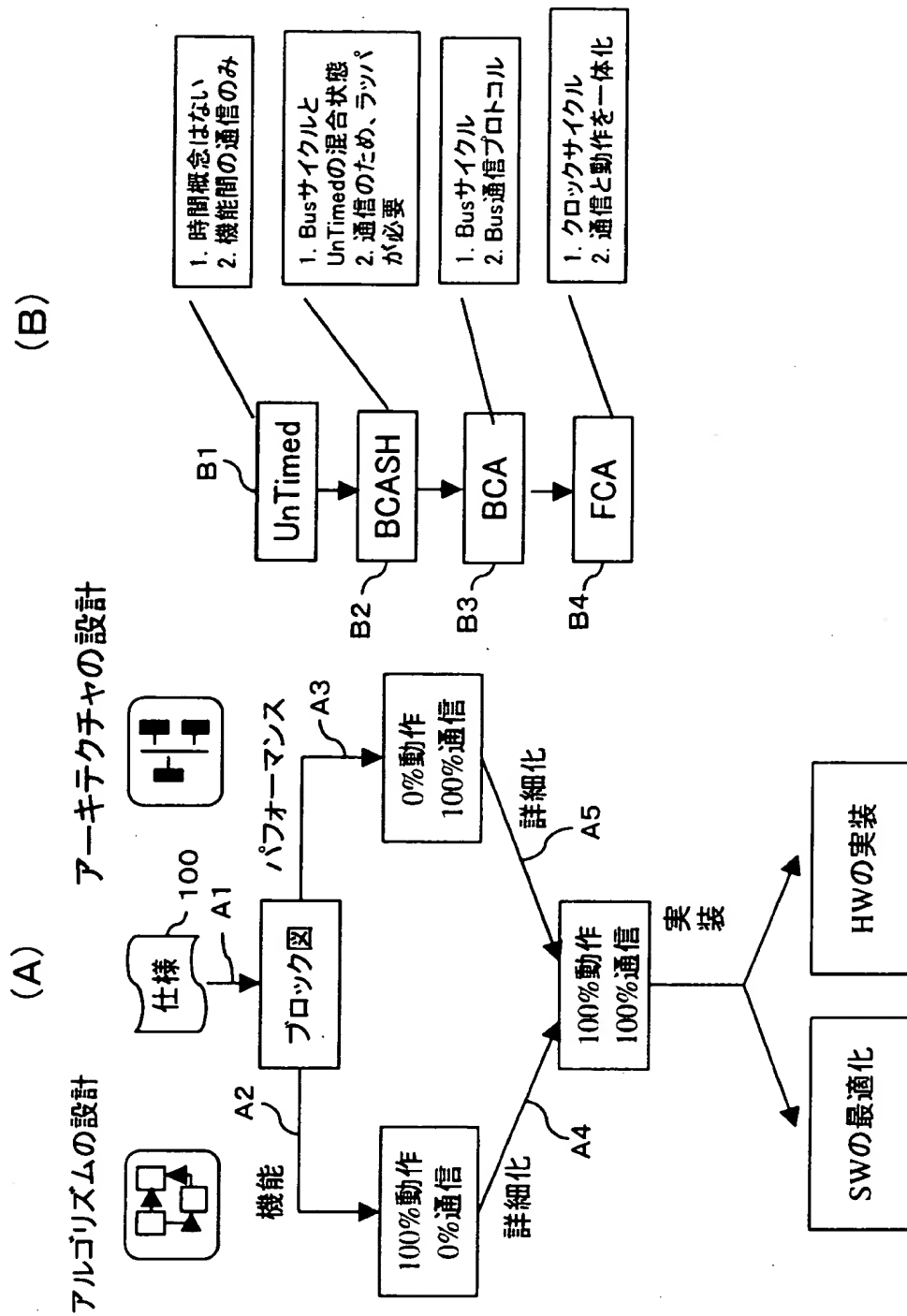
31c'

313'

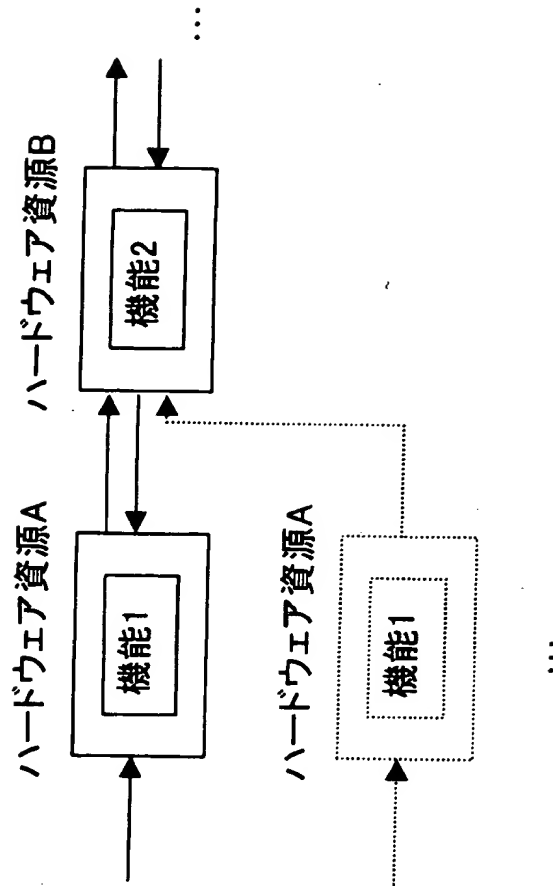
【図29】



【図30】



【図 3 1】



【書類名】 要約書

【要約】

【課題】 論理装置の設計の初期段階において、動作シミュレーションを行なうて、機能の確認、アーキテクチャの評価を行なえるようにするとともに、アーキテクチャの変更に対しても最小限の記述の変更で柔軟に対応できるようにする。

【解決手段】 スレッドマネージャ 1 1 が、論理装置の動作完了までに必要となる機能を表したスレッド 1 3 の実行に必要なハードウェアリソース 1 4 をリソースマネージャ 1 2 が要求に応じて割り当て、さらに、スレッドマネージャ 1 1 がその割り当て結果に応じてスレッド 1 3 の実行状態を制御するとともに、スレッド 1 3 の実行が完了するまでスレッドマネージャ 1 1 とリソースマネージャ 1 2 とが連携してリソース要求、リソース割り当て及びスレッド制御を繰り返し実行することにより、論理装置の動作完了までの動作をシミュレーションする。

【選択図】 図 3

出 願 人 履 歴 情 報

識別番号 [000005223]

1. 変更年月日 1996年 3月26日

[変更理由] 住所変更

住 所 神奈川県川崎市中原区上小田中4丁目1番1号

氏 名 富士通株式会社